A MODEL OF DYNAMIC COMPILATION FOR HETEROGENEOUS COMPUTE PLATFORMS

A Thesis
Presented to
The Academic Faculty

by

Andrew Kerr

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2013

A MODEL OF DYNAMIC COMPILATION FOR HETEROGENEOUS COMPUTE PLATFORMS

Approved by:

Dr. Sudhakar Yalamanchili, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Jeff Shama
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Mark Richards
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Aaron Lanterman
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Santosh Pande
School of Computer Science
Georgia Institute of Technology

Date Approved: November 19, 2012

To the students of mathematics, science, and engineering.

ACKNOWLEDGEMENTS

I am greatly indebted to my adviser, Sudhakar Yalamanchili, whose insights, wisdom, and technical breadth shaped my understanding of computer engineering and role as a scholar. Mark Richards sets a high standard for excellence and deserves thanks for helping me pursue my interests early in graduate school and has contributed greatly to the completion of this thesis. Santosh Pande provided an initial understanding of research topics in compilation and shared numerous insights in applicability of this work. Aaron Lanterman's unique teaching style illuminated the concepts of signal processing and abstract algebra during several points in my career as a student. He is also responsible for encouraging my monomaniacal pursuit of GPU computing. I am also thankful for Jeff Shama's role on my committee, as a welcome critic of this work.

There have been several other individuals who have had a strong impact on my research interests, who have enabled career opportunities and contributed valuable ideas, feedback, and criticism. Dan Campbell initiated my early exploration of GPU computing, guided the framing of interesting research questions, and has offered tremendous insights on computing, research, and professional development during my time at Georgia Tech. Hyesoon Kim's vision and perspective has shaped development, application, and popularization of this work. Vinod Grover offered me the opportunity to work on compilers in a production environment and continues to be a great source of provocative discussion. Nate Clark instilled a solid foundation in dynamic compilation, and his high expectations provided the impetus for this research topic.

This thesis was completed in spite of Emma Loggins's wonderful and endearing presence in my life, who provided me with countless reasons to leave work early each day, yet understood when I occasionally declined. Without her encouragement and excellent example, many of my goals may not have come to fruition.

I will gladly remember the triumphant moments, the challenges, and contagious enthusiasm among the students of the Computer Architecture and Systems Lab, the Center for Experimental Research in Computer Systems, and the contributors to the GPU Ocelot project. Each interaction of ours has imparted a unique and favorable perturbation to my perspective and will be thoroughly cherished. I wish to acknowledge in particular Greg Diamos, Jeff Young, Naila Farooqui, Eric Anger, Nawaf Almoosa, Chad Kersey, Haicheng Wu, Rodrigo Dominguez, Si Li, Tri Pho, Aswhin Lele, Alex Merritt, Mukil Kesavan, Mitchelle Rasquinha, Druhv Choudhary, William Song, Minhaj Hassan, and Vishakha Gupta.

I would also like to thank several individuals I am immensely fortunate to know. Joseph Iacobucci, Nicolas Bunzmann, and Mike Faria have been excellent friends and great sources of inspiration regarding critical thinking, effective problem solving, and healthy self-interest. They have been endless sources of good advice; I occasionally wish I had taken some of it. I also wish to acknowledge Josh, for quitting college, joining start-ups, growing into a first-rate innovator and technical leader, and helping to grow his company into a billion dollar enterprise. Thanks, specifically, for withholding his pity for me as his successes were richly and justly rewarded.

Finally, I am exceedingly thankful for the unwaivering encouragement, love, and support from my parents, Robert and Lynda, and my sister, Caitlyn, who has become a fine and diligent scholar. They have been a grand foundation to which I owe my accomplishments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Summary

Trends in computer engineering place renewed emphasis on increasing parallelism and heterogeneity. The rise of parallelism adds an additional dimension to the challenge of portability, as different processors support different notions of parallelism, whether vector parallelism executing in a few threads on multicore CPUs or large-scale thread hierarchies on GPUs. Thus, software experiences obstacles to portability and efficient execution beyond differences in instruction sets; rather, the underlying execution models of radically different architectures may not be compatible. Dynamic compilation applied to data-parallel heterogeneous architectures presents an abstraction layer decoupling program representations from optimized binaries, thus enabling portability without encumbering performance. This dissertation proposes several techniques that extend dynamic compilation to data-parallel execution models. These contributions include:

- Metrics for characterizing performance of GPU computing workloads
- Statistical performance modeling of heterogeneous workloads
- Execution model translation for vector processors
- Region-based compilation and scheduling
- Prototype heterogeneous compiler implementation and critique

We evaluate these claims via the development of a novel dynamic compilation framework, GPU Ocelot, with which we execute real-world workloads from GPU computing. This enables the execution of GPU computing workloads to run efficiently on multicore CPUs, GPUs, and a functional simulator. We show data-parallel workloads exhibit performance scaling, take advantage of vector instruction set extensions, and effectively exploit data locality via scheduling which attempts to maximize control locality.

CHAPTER I

INTRODUCTION

The computing industry is experiencing a shift in which clock frequency is no longer a viable method for performance scaling, and processor designs are instead leveraging parallelism as a path toward innovation. Due to slowing clock frequency scaling and rising power consumption, the advance of Moore's Law yields growing numbers of transistors but little improvement in processor critical paths. Thus, emerging processor designs are placing greater emphasis on heterogeneous processors. General-purpose Graphics Processor Units (GPUs) and wide on-chip vector units in the case of Advanced Vector Extensions (AVX) are commonplace and becoming more crucial in delivering faster machines.

Systems composed of these types of architectures have demonstrated increases in peak throughput and power efficiency, yet their complexity presents challenges toward programmer productivity and performance portability. To utilize distinct processors, multiple versions of a program are needed. Frequently, these must be hand-tuned to the target hardware which may become wasted effort if the target execution environment consists of different combinations of processors. Additionally, the rise of parallelism adds an additional dimension to the challenge of portability, as different processors support different notions of parallelism, whether vector parallelism executing in a few threads on multicore Central Processing Units (CPUs) or large-scale thread hierarchies on GPUs.

To accommodate vastly different architectures in the face of widespread parallelism, an abstraction layer is needed to transform the representation of a program into a form that is efficiently executable on available processors. This abstraction layer should detect which parts of a program are likely to take advantage of specialized hardware, and it should be capable of generating efficient

executable code for that hardware given the nature of concurrency it exposes.

1.1 Contributions

Thesis: *This research argues that dynamic compilation from a data-parallel execution model efficiently utilizes heterogeneous compute systems composed of parallel processors.* This work demonstrates that a dynamic compilation and managed runtime environment yields a platform on which one data-parallel program representation can be an effective way of developing applications for execution on heterogeneous compute platforms. Through dynamic compilation, these applications are **portable** across systems with different types of CPUs and GPUs with varying levels of parallelism and instruction set architecture (ISA) features. Execution performance is **scalable** with respect to workload size and system-wide throughput capacity. Both of these goals are achieved without encumbering the programmer with the need to supply multiple implementations of each major computation for the set of available processor architectures. Thus, **productivity** is not diminished in the face of heterogeneity.

This research examines prevailing commodity heterogeneous processors, identifies characteristics of common workloads, and presents GPU Ocelot, a novel dynamic compilation framework for evaluating compilation and online optimization techniques for heterogenous computing. With the Ocelot framework, this research has developed novel execution model transformations and dynamic compilation techniques in the context of explicitly data-parallel kernel-oriented workloads. This model of dynamic compilation makes the following contributions.

Metrics for Characterizing Performance of GPU workloads. GPU programming models emphasize parallelism, uniformity of control flow, coordinated access to shared data structures, and careful consideration of data flow within an application. This thesis defines several metrics derived from the PTX abstract machine model to measure efficiency of execution, utilization of compute

resources, data flow among threads, and parallelism scalability. We apply these metrics to off-the-shelf GPU computing applications and present recommendations for application developers and GPU architects. Additionally, we draw several conclusions about current limitations in GPU programming models based on case studies in implementing and optimizing dense linear algebra computations.

Statistical Performance Modeling. Heterogeneous computing necessarily presents challenges related to allocating compute resources for a given applications and tasks. We define an automated statistical approach to modeling application runtimes and other figures of merit as machine parameters are varied to provide feedback for task-level scheduling decisions to optimize application throughput on heterogeneous platforms. Profile-driven scheduling is readily applied to additional problem domains such as distributed large-scale simulation. We evaluate the proposed technique via a prototype implementation utilized for rapid design-space exploration.

Execution Model Translation. Heterogeneous computing exacerbates demands for application portability. Systems composed of radically differing processors present incompatibilities in terms of instruction set architecture and, more critically, translating elements of the source execution model to efficiently utilize hardware resources. Concretely, in the context of parallel computing, this implies a reduction of concurrency from maximally available parallelism to available hardware parallelism in target processors. The bulk of this thesis follows from the counterintuitive notion that compiler-managed *serialization* of parallel programs is critically important to their efficient execution, once appropriate parallel algorithms have been selected and implemented in a suitably expressive program representation. We explore techniques for parallelism reduction and introduce a novel approach to **vectorization** in the presence of control flow, achieving speedups of 45% on today's workloads on commodity processors. This work is motivated by roadmaps from microprocessor designers that focus on enhancing CPU performance by coupling scalar datapaths to progressively wider SIMD functional units. We expect the techniques proposed and implementation evaluated in this thesis to be directly applicable to these future processor architectures.

Region-based Compilation and Scheduling. Dynamic compilation necessitates inescapable energy, runtime, and storage overheads during application runtime. Aggressive specialization attempts to optimize special cases of application execution which exacerbates overheads if multiple specializations are needed. Independently, we observe that phase behaviors vary within regions of data-parallel kernels, and that performance and energy improvements are possible when threads executing the same region of a program are executed consecutively or concurrently. We present a solution to region-based program optimziation and thread scheduling that simultaneously reduces dynamic compilation overheads while realizing performance gains through more frequent thread scheduling. We demonstrate this technique in the context of execution model translation but describe follow on work such as how this technique may be used to implement nested parallelism and execution preemption on GPUs.

Prototype Heterogeneous Compiler. This work was evaluated in the context of GPU Ocelot, a novel dynamic compilation framework developed specifically to target data-parallel programming models. In this thesis, we describe design decisions and novel implementation details that may be of particular interest to developers of future retargetable dynamic compilation frameworks for heterogeneous platforms. We describe an analysis and optimization pipeline for data-parallel kernels, an interface for user-defined instrumentation and profiling tools, details for translating instruction sets to different processor architectures, and productivity tools developed to aid the development and optimization of GPU compute applications. We also provide criticism of these decisions in light of recent applications and a full-level performance evaluation.

1.2 Organization

This thesis is organized as follows.

Chapter 2 discusses several mainstream heterogeneous processors including NVIDIA “Fermi”

GPU, Intel SandyBridge, and AMD Fusion CPU architectures. This section also diccusses heterogeneity at the system and processor levels, with task-level heterogeneity targeting different processors and instruction-level heterogeneity targeting vastily different functional units within the same processor pipeline.

Chapter 3 describes the prototype implementation of several computationally intensive kernels including QR decomposition. These computations are exposed via a standardized API for linear algebra and reveal several challenges associated with library-based approaches to drive the proliferation of heterogeneous computing with GPUs. This work then defines several critical metrics characterizing data-parallel workloads and evaluates these metrics against a broad set of GPU computing benchmarks. Optimization strategies are developed and discussed, and a novel thread reconvergence mechanism is presented.

Chapter 4 presents a novel alternative to traditional simulation-based computer architecture research: statistical performance modeling. Applications are profiled using heavy-weight simulators executing representative workloads, and the resulting profiles are used to compute regression models of performance, including runtime and energy consumption. These are then applied to future workloads to estimate performance and enable tuning of architecture parameters. Predicting performance for heterogeneous workloads enables a runtime to make resource management decisions. Determining which of a set of available processors to execute the next task is a very useful decision, enabling a runtime to compute an optimal task schedule.

Chapter 5 describes techniques for compiling data-parallel kernels for efficient execution on multicore heterogeneous CPUs. This addresses the problem of thread fusion in which explicitly independent threads are serialized. Thread fusion enables software to express the maximum amount of parallelism within a computation and permits the compilation framework and hardware to satisfy this parallelism with hardware resources available. Thread fusion is extended with a novel program transformation to target Single-Instruction Multiple-Data (SIMD) functional units which are increasingly common sources of performance improvements in modern CPUs. This

technique is ISA agnostic and tolerant of control-flow divergence without predicated serialization.

Chapter 6 describes program transformations for kernel-oriented programming models that improve efficiency on heterogeneous platforms. Dynamic compilation frameworks are uniquely poised to split, fuse, and transform kernels as they are executing based on frequently exhibited behaviors, even if they are data-dependent. This work extends the Just-in-Time (JIT) compilation concepts applied for execution model translation to partitioning whole programs and scheduling their execution in pursuit of several optimizations to increase control-flow uniformity, increase cache locality, and reduce scheduling overheads.

Chapter 7 describes the engineering design and implementation decisions made during the development of GPU Ocelot. At the time of this writing, GPU Ocelot is the only heterogeneous compilation framework that uniformly targets NVIDIA GPUs, multicore CPUs with scalar and SIMD data paths, and AMD Radeon GPUs. Additionally, GPU Ocelot includes an emulated execution target for prototyping novel architectural and software features which can be used to drive an external cycle-accurate GPU simulator with instruction traces from real workloads. Ocelot's immense flexibility has enabled considerable research and productivity at Georgia Institute of Technology, Northeastern University, and research collaborations with NVIDIA and AMD.

Chapter 8 presents lessons learned from this research and comments on future investigations and applications of dynamic compilation on heterogenous and data-parallel architectures.

CHAPTER II

HETEROGENEITY

Heterogeneity in computer engineering refers to systems composed of multiple processing elements with fundamental differences. These may include instruction-set heterogeneity in which processing elements utilize distinct and non-overlapping instruction set architectures (ISAs) and must consequently be programmed using distinct toolchains. Alternatively, processors may have performance asymmetry in which processors with compatible ISAs contain different sized caches or have fewer functional units. Programs will still execute correctly on each processor but exhibit different performance characteristics such as faster runtimes or greater energy efficiency. Heterogeneity presents the problem of efficiently matching programs to processors, as compute-bound programs may benefit from additional functional units whereas programs with irregular memory behavior may benefit from large caches or sophisticated hardware prefetching. Functional asymmetry refers to systems with high overlap in instruction sets but with some processors possessing certain enhancements such as vector functional units, application-specific accelerators, or entropy generators. High overlap in ISA compability enables lighter weight schemes such as emulation to address functional differences among processors thereby accommodating the needs of software. Finally, heterogeneity can refer to execution model asymmetry in which a system utilizes multiple distinct execution models, and programs must be radically transformed to execute correctly when moving from one processor to another. Broadly, heterogeneity addresses the growing physical obstacles to delivering high performance from a single general-purpose processor.

2.1 Rise of Heterogeneous Computing

Innovation in computing has historically been driven by increasing performance of microprocessors, in large part derived from improvements in semiconductor fabrication technology. Dennard scaling [46] projects decreasing MOSFET channel length with each successive generation enabling linearly increasing clock speed, reductions in voltage, and near constant per-device power consumption. With higher levels of integration, additional architectural features that exploit parallelism among consecutive instructions enabled further improvements in performance. Not only were clock rates increasing, but the number of instructions that could be retired per clock cycle (IPC) also grew. Superscalar processors issuing multiple instructions per cycle has been a widely successful approach to improving system performance and microarchitectural enhancements were opaque with respect to software. Legacy applications could take advantage of innovations simply by running them on new hardware thus fueling a tremendous amoung of economic growth in the computer software industry. This strategy has been effective, and processors with this design have dominated laptop, desktop, and server markets for nearly two decades. Yet, computer architects are facing several limitations to traditional approaches. Power consumption, instruction-level and thread-level parallelism limits, and interconnect delay are conspiring to force radical changes in both processor design and programming models.

Dynamic power consumption in a digital circuit is proportional to clock frequency and the square of operating voltage according to the well-known CMOS power model $P = \alpha C V_{dd}^2 f$. As feature size has decreased, decreases in operating voltage have been accompanied by increases in clock frequencies. Today, a processor operating at over 3 GHz is commonplace among medium-performance desktop workstations and high-end portable machines. Coupled with high levels of integration, power density has increased dramatically. Modern CPUs typically exhibit over 100 W of nominal dissipated power. This approaches the upper bound of what traditional cooling systems can accommodate, and the proliferation of mobile devices with limited battery life is placing a

greater emphasis on low-power compute systems. Consequently, the area density of dissipated heat has emerged as a practical limit to clock frequency scaling.

To mitigate the slowing of clock frequency scaling, processor designers have resorted to increasing the number of processor cores per die. This architectural trend exploits thread-level parallelism and has enabled the continued growth in processor performance. However, a study by Esmaeilzadeh, et al. [54] examining combinations of large and small, in-order cores implemented in a hypothetical 8 nm technology node evaluated with the PARSEC [12] benchmark suite shows asymptotic performance bounds even as cores are added. With an unrealistically high 500 W power budget, the dearth of task-level parallelism in traditional workloads limits the utility of additional cores. Other types of of parallel execution models (such as CUDA's data-parallel kernels) exhibit tremendously more parallelism that may enable continued performance scaling with additional data paths and no unutilized "dark silicon" as the authors suggest. Yet, this requires a significant departure from traditional programming models.

Two-dimensional feature scaling reduces transistor area and gate delay but increases signaling delay due to wires' decreasing cross sectional area. Consequently, signaling latencies of on-chip networks for sub-micron semiconductor processes are increasingly limited, and single-cycle global communications are impossible for even moderately sized dies. In practice, this limits the area complexity of single cores [55] and encourages small data structures to limit data movement [158] and energy consumption [88]. Towles [43] describes a simple on-chip interconnection network with modest area requirements that avoids the drawbacks of global wiring networks. Highly parallel tiled architectures such as Tilera's Tile64 [40] and Intel's Cloud On Chip [103] attempt to overcome wire delay limitations by arranging tens of processor cores in a mesh with the on-chip network explicitly exposed to the programming model.

The International Technology Roadmap for Semiconductors indicates transistor density will continue to increase, peaking in a planned 11 nm technology node by 2015 [85]. Smaller devices yield greater levels of integration, yet this poses an organizational question of how to achieve

throughput and efficiency gains with more transistors while respecting contraints in power, clock frequency, and interconnect delay. Hameed et al. [70] observe ASICs can be over 500x more energy efficient than Chip Multiprocessors (CMPs) for throughput-oriented tasks. A microcosm of the industry shift toward heterogeneity, their work describes a series of transformations applied to a soft-core processor transitioning it from a general-purpose programmable CMP toward an application-specific processor.

2.2 Heterogeneous Manycore

Composing a compute platform of a collection of highly efficient but specialized processors has become the plan of attack for heterogeneous manycore computing. Limits to traditional architectures have motivated the exploration of accelerators, domain-specific processors, and parallelism on a massive scale. These efforts incur significant increases in software complexity as programs must somehow use each type of processor efficiently. This section focuses on the features of modern CPU architectures that are distinctly heterogeneous.

Chip Multiprocessors (CMPs) integrate several microprocessors on the same die, typically sharing a last-level cache, inteconnect, and access to off-chip memory. CMPs are a straightforward reaction to nascent scalability limitations affecting uniprocessors; large transistor budgets are applied to greater numbers of cores. Most desktop processors shipping today include two to four cores, and many mobile devices are beginning to ship with multicore CPUs. Today's CMPs also exhibit heterogeneity by including SIMD instruction set extensions for throughput processing and other specialized functional units. Moreover, some research points to asymmetric CMPs [147] in which more die area is allocated to a latency-oriented out-of-order core, while the other cores in the CMP are simple, low-area in-order cores. This arrangement enables the large core to execute latency-sensitive threads while the other cores execute throughput-oriented workloads.

Graphics Processing Units [101] are commodity accelerator architectures traditionally used to perform rasterization of 3D graphics accessed by APIs such as DirectX and OpenGL. These exploit high levels of parallelism and latency tolerance in rasterization algorithms and achieve high throughputs with moderate clock speeds and large numbers of simple, in-order cores when compared to traditional CPU microarchitectures. Economies of scale have resulted in GPUs capable of 2-5x higher peak floating-point performance than similarly-priced CPUs. Modern GPUs are capable of executing programs containing control flow and arbitrary scatter and gather operations making them as programmable as CPUs. Consequently, a large and growing body of research has been initiated to understand how to map high-value computations onto GPUs.

This research focuses on GPU computing, so it is worth understanding GPU architectures in some detail. Figure 1 provides an overview of a modern GPU architecture. Streaming Multiprocessors (SMs) are analogous to "cores" in the vernacular of multicore CPUs. Kernels are written to spawn a large number of threads which are then oversubscribed to SMs. While modern CPUs typically devote a large fraction of die area to caches (8MB or more for Intel Nehalem quad-core CPUs, for example) to alleviate latencies of the memory hierarchy, GPUs have very small caches and devote more area to register files, functional units, and interconnects. NVIDIA's "Fermi"-class flagship GPU contains four Graphics Processor Clusters, each partitioned into four SMs. Each SM contains 64KB of high-speed memory partitioned into an L1 cache and a scratchpad. SMs contain 32 stream processors, instruction issue and thread scheduling logic, and a 128 KB register file. The large register file and relatively small caches place a great emphasis on maintaining the live state of hundreds of thread contexts. High latency to off-chip memory is hidden by scheduling other threads, and programs focus on high throughput workloads rather than low-latency computations.

Systems containing GPUs are, by definition, heterogeneous. With the exception of the aborted Intel Larrabee processor [136], GPUs utilize instruction set architectures distinct from the host system's central processor and typically do not run an operating system. Applications are written

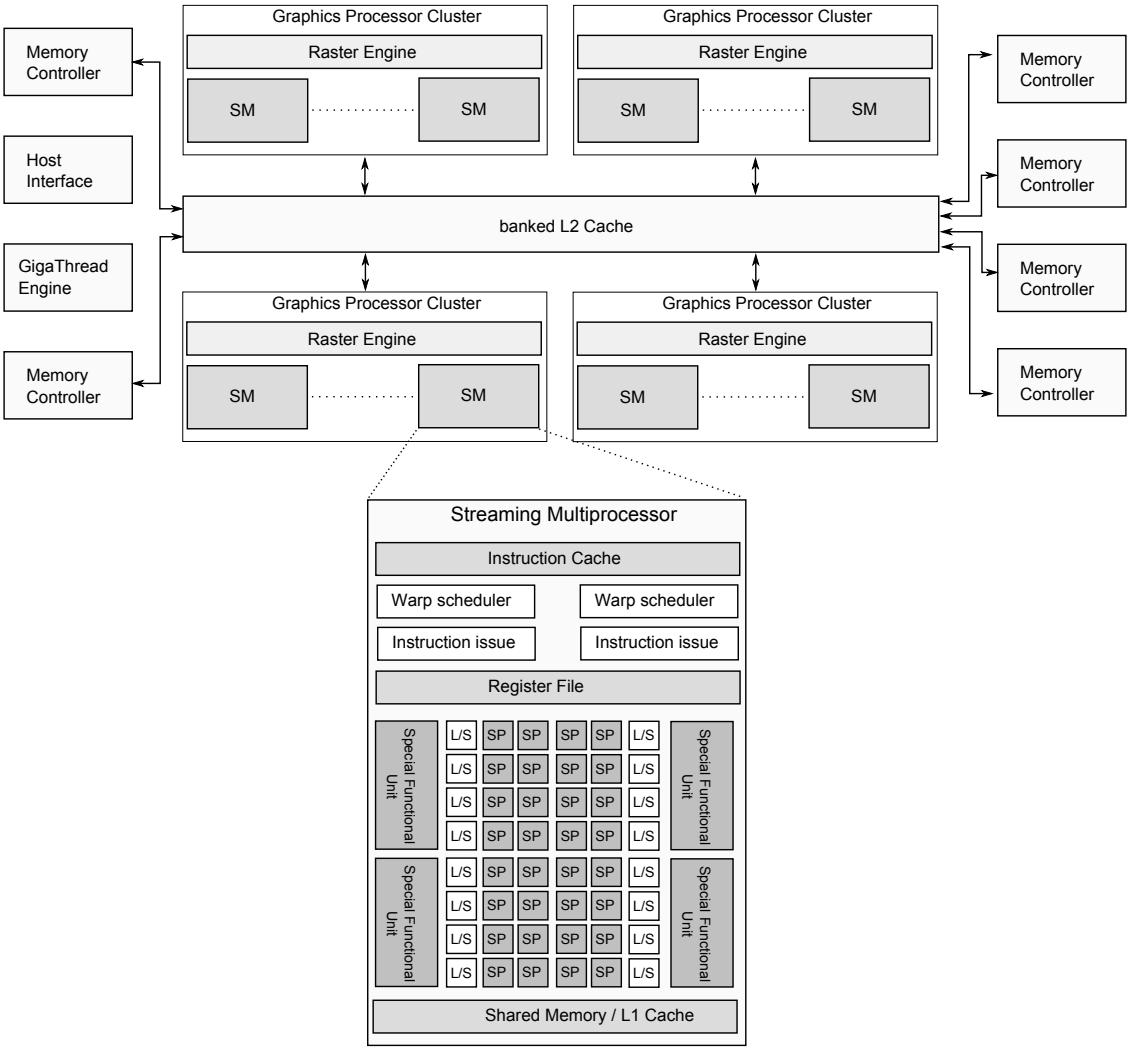


Figure 1: Block diagram of NVIDIA GF100 "Fermi" GPU architecture.

in terms of data-parallel kernels which are launched by an application running on the host processor. These computations are performed by attached GPU devices and operate on data structures resident in GPU memory, typically distinct from system memory. Utilizing GPUs and CPUs in the same system is an open research challenge.

Heterogeneous processors include multiple distinct architectural features in the same multiprocessor. The Cell Broadband Engine [87] contains an out-of-order symmetric multiprocessor POWER

Table 1: Characteristics of modern heterogeneous processors.

	NVIDIA GTX9800	NVIDIA GTX280	NVIDIA GTX590	Intel Core2 Q9550	Intel Core-i7
ISA	G92	GT200	GF104	x86-64	x86-64
Clock Speed (MHz)	1500	1296	1215	2830	2600
Cores	16	32	15	4	4
Hardware Threads per Core	352	384	576	1	2
Issue Width	2	2	3	3	4
Vector Width	8	2	32	4	8
Peak FLOPs (GFLOP/s)	576	933	1244	108	166
Memory Bandwidth (GB/s)	64.0	141	164	8.0	18.4

core for traditional workloads as well as eight in-order accelerator cores known as Synergistic Processing Elements (SPEs). SPEs contain four-wide vector datapaths and a software-managed scratchpad memory in lieu of a coherent L1 cache. SPEs do not access main memory, but rather a programmable DMA engine initiates transfers between SPE scratchpad memory and off-chip system memory. Modern CPUs such as Intel's Sandybridge [89] architecture and AMD's Bulldozer [23] are presently shipping with multicore CPUs and GPUs (with distinct ISAs) integrated on the same die. Like the Cell processor, integrated CPU-GPU pairings require two different programming models and distinct compilation toolchains to utilize the entire die, thus incurring a significant increase in software complexity. Table 1 lists performance characteristics of modern commodity GPUs and CPUs, illustrating the dichotomy between throughput- and latency-oriented design goals.

2.3 Data Parallelism

Data parallelism describes a model for parallel computing in which a set of operations are broadcast and executed concurrently over large sets of data. This contrasts with multithreading in which tasks are mapped onto threads which then execute in parallel. This distinction has given rise to a body of work describing efficient algorithms [76] for data parallel computation that may be efficiently realized by vector processors. This distinction makes programming highly parallel systems

with tens or hundreds of processors tractable, as the same algorithm implementation may be used when data set size and the number of processors grow. Blelloch [15] defines several algorithmic primitives such as the segmented scan which have inspired high-performance implementations of common tasks on modern processor architectures such as a fast in-memory radix sort [117].

NVIDIA's Compute Unified Device Architecture (CUDA) [124] is a programming language and API for executing programs on GPUs without casting computations in terms of 3D rasterization operations. Computations intended to execute on the device are expressed as data-parallel *kernels* which are then launched by the host application over thousands of light-weight GPU threads. The CUDA Runtime API [126] is an application programming interface for orchestrating kernel launches, managing GPU device memory allocations, and loading additional program binaries on the GPU.

Illustrated in Figure 2, the CUDA execution model semantics describe a tiered hierarchy of threads. At the lowest level, collections of threads are mapped to a single *stream multiprocessor* (SM) and executed concurrently. This collection of threads is known as a *cooperative thread array* (CTA), and kernels are typically launched with tens or hundreds of CTAs which are oversubscribed to the set of available SMs. The programming model does not define global synchronization between SMs except on kernel boundaries, and the mapping of CTAs to SMs is undefined. In [49], we have argued CUDA is a realization of the bulk-synchronous parallelism model described by Valiant [151] due to constrained synchronization domains and weak consistency guarantees of the memory hierarchy.

In the wake of CUDA's popularity, a consortium was formed to create a vendor-neutral alternative and thus OpenCL [66] (Open Compute Layer) was born. OpenCL borrows many of the concepts from CUDA such as the two-tiered thread hierarchy and kernel-oriented programming model. An alternative API was designed that largely resembles the CUDA Driver API to the extent that many CUDA applications can be ported to OpenCL without requiring modifications at the algorithmic level. Together, CUDA and OpenCL provide similar kernel-oriented programming

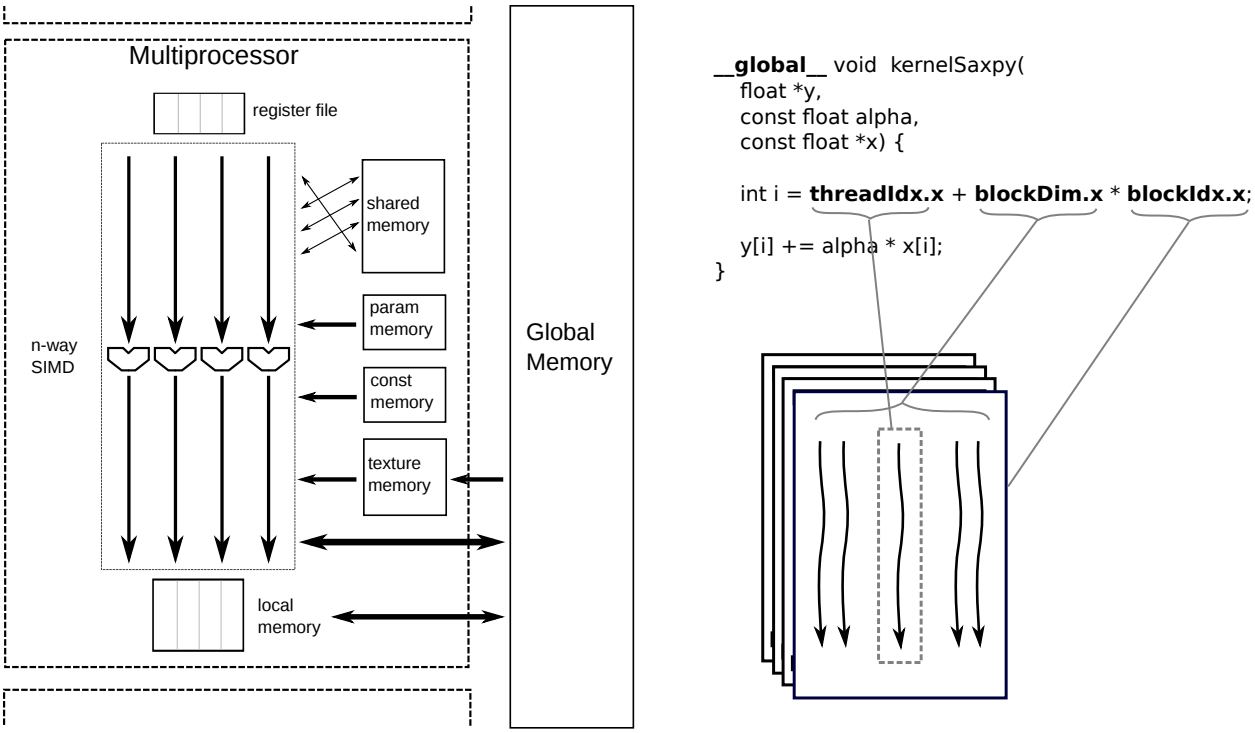


Figure 2: CUDA thread hierarchy and abstract machine model.

models with kernels consisting of tiered sets of threads exhibiting fine-grain and coarse-grain synchronization semantics.

In the six years that CUDA has been available, thousands of applications and numerous libraries [77, 93, 150], have been written that take advantage of commodity pricing and high performance of GPUs, making it a very popular programming model for heterogeneous computing. CUDA exposes parallelism explicitly as well as other architectural features of the underlying hardware such as software-managed scratchpad memory. This low-level exposure of hardware features coupled to powerful features of C++ such as templates creates a spectrum of program quality with high performance on one end and succinctness on the other. At the time this work began, CUDA was the most flexible production-quality programming model and toolchain available. Since its release, CUDA has remained popular among GPU computing developers, and this popularity has availed researchers with numerous real-world applications on which to evalute their contributions.

The contributions of this work have been implemented with CUDA workloads in mind, and this decision does not appear to be a drawback.

2.4 Dynamic Compilation

The fundamental role of a compiler is to facilitate programming language abstractions that reduce software development costs and increase software flexibility. Consequently, the compiler is a vital component in the effort to accommodate machine diversity. Compilation tools have been the focus of research intending to exploit ISA-specific machine enhancements, enable high-level language portability, and ultimately become part of the execution environment. As computing resources become more diverse, compilation tools are likely to become even more crucial in obtaining efficiency and performance while maintaining portability and flexibility.

2.4.1 Modern Dynamic Compilers

Dynamic compilation [28] is a step toward decoupling a portable representation of a program from a machine-specific binary that may execute it by compiling it just before it is run. Static compilation applies conservative optimization decisions that must be suitable for all execution targets and may miss out on particular capabilities of some target machines in exchange for generality. On the other hand, dynamic compilation makes optimization choices with knowledge of a program's actual execution environment.

Early examples such as the FX!32 [27] dynamic binary translator and IBM Daisy [52] provide ISA compatibility to new processor instruction set architectures under the belief they would be significantly more efficient than legacy Complex Instruction Set (CISC) ISAs. FX!32 combines emulation and profiling with binary translation to provide x86 emulation on DEC Alpha CPUs [35] whose RISC architecture and high clock speeds exceeded the performance of comparable x86-based CPUs. The FX!32 virtual machine emulates sequences of x86 instructions in software

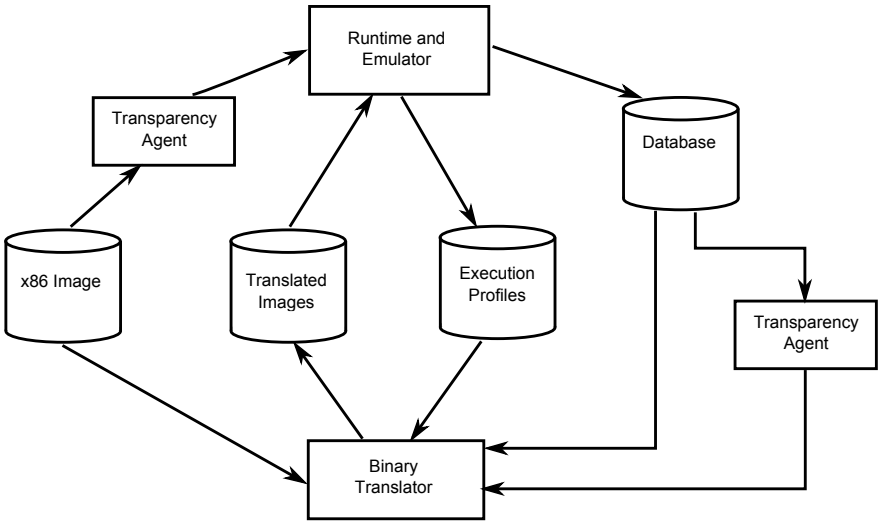


Figure 3: Block diagram of FX!32 emulation and binary translation environment. FX!32 performs emulation, profiling, and binary translation of x86 guest applications running on an Alpha CPU.

while constructing an execution profile, and frequently encountered blocks are dispatched to a background translation task which translates source basic blocks into the Alpha instruction set during idle periods. These translations are cached, and the next time these blocks are encountered during emulation, control jumps into the code cache for higher performance execution in the native ISA. Transitions from the emulator to the code cache require restoring the state of architectural registers and stack frames followed by an indirect jump. A block diagram of FX!32 appears in Figure 3, and this model has been the basis for numerous JIT compilers and virtual machines to follow.

IBM Daisy [52] implements a similar form of binary translation from the PowerPC ISA to that of a custom Very Long Instruction Word (VLIW) processor. Unlike out-of-order superscalar processors, VLIW processors [58] depend on compilation to identify instruction-level parallelism and construct efficient code schedules that exploit this ILP in hardware. In a sense, the dynamic instruction scheduling hardware in superscalar processors is replaced by a compiler. This critical dependency on static code schedules matches capabilities afforded by dynamic compilation well,

as optimizations may be performed in light of runtime behavior and applied on instruction traces and across function calls. Daisy achieved on average 2.5 PowerPC instructions per very long instruction word and avoided much of the complexity of an out-of-order superscalar CPU.

Transmetta extends this concept in hardware with their Crusoe microprocessor [45] which performs lightweight binary translation online, concealing the processor's native ISA entirely. Transmetta's Crusoe combines a VLIW processor with a software layer that interprets x86 instruction streams, performs online dynamic binary translation and optimization, and integrates a runtime that manages the execution of translations. Crusoe relies extensively on hardware features supporting dynamic translation, and the VLIW architecure executes x86 workloads with high energy efficiency. Unlike IBM Daisy, Crusoe simulates the x86 instruction set at the processor level and cannot rely on operating system or application support; it must even run x86 BIOS. The underlying VLIW processor includes extensive support for speculative execution, shadowing x86 architectural registers, and performing aggressive optimizations in software such as trace scheduling [34], moving instructions across basic blocks [139], and memory reordering [60].

The success and proliferation of Java is due to advances in virtual machines and dynamic compilation. Early Java virtual machines relied on interpretation to provide the managed execution environment defined by the Java Virtual Machine Specification, but subsequent developments in just-in-time compilation [4] vastly improved the performance of Java applications without compromising Java's design goals of portability and safety. Infrastructures such as Dynamo [10] and Jalapeño [22] integrate compilers with virtual machines for online compilation. In Jalapeño, all procedures are compiled before they are executed with lightweight optimizations enabled; frequently executed regions are then subject to more expensive and aggressive optimization passes. The .NET Common Language Infrastructure (CLI) [115] extends the concept of an execution manager hosting a platform-independent virtual instruction set while maintaining interoperability with unsafe language features such as access to native hardware and unmanaged pointers.

Low Level Virtual Machine (LLVM) [105] presents a unified compilation infrastructure for

interprocedural analysis and online just-in-time compilation. Entire applications are compiled to a common internal representation (IR) from various language front ends such as C, C++, and Fortran. This IR facilitates data-flow analysis, a strict type system, structured accesses to memory, and exception handling. LLVM supports serializing program representations, idle-time optimization, profiling, inter-procedural analysis and optimization, and does not enforce a particular runtime model unlike Java and the .NET CLR. LLVM decouples language front ends, optimization on its intermediate-level IR, and machine-specific backends. These properties make LLVM a productive starting point for custom compilers and virtual machines. Indeed, the GPU Ocelot [48] framework described extensively in this thesis targets LLVM in the implementation of its multicore CPU backend.

2.4.2 Feedback-Directed Optimization

Feedback-directed optimization (FDO) [138] refers to altering a program at runtime in response to observed behavior. Adve et al. [5] describe a continuum of optimizations from conservative decisions made with results of static analysis on one end of the spectrum to optimizations with complete knowledge of runtime behavior on the other end. Optimization in response to frequent application behaviors is a common motif throughout computer architecture. Caches, translation look-aside buffers, and branch predictors all attempt to improve performance over conservative baseline cases based on previously encountered behaviors. However, performing this type of feedback-directed optimization in hardware requires die area, increases processor complexity, and consumes significant fractions of the total power budget.

Beyond hardware implementations, FDO may be realized by off-line compilation in response to application profiles. This type of Profile-Guided Compilation (PGC) relies on instrumented executions of applications running representative data sets selected by developers before the application is shipped. PGO has yielded average speedups of 17% for the SPECint95 benchmarks on an Alpha

processor [31] compared to a baseline static compilation using the most aggressive optimizations. The authors attribute performance gains to aggressive inlining [145], superblock formation [74], and loop restructuring [111]. Moreover, the authors find that opportunities exploited by PGO increase with program complexity, particularly when inter-procedural optimizations are possible.

PGO works well when the training data sets closely resemble real-world workloads, and target machine parameters resemble the training machines. Applying optimizations nearer to an application's execution enables more accurate profiling. *Dynamic optimization* refers to applying optimizations as the program is executing by the end user. Zhang et al. [160] describe an automated process in which profiles were gathered by a background process, and optimizations performed during idle periods within an application to minimize the compiler's impact on runtime. To be effective, this type of dynamic optimization must avoid compilation overheads unless (1) the expected impact of the optimization is high and (2) analysis and optimizations are applied to regions particularly amenable to optimizations. IBM DAISY [52] and IBM Jalapeno [22] apply a low-overhead optimization pass over all code regions then, in light of profiling information, aggressively optimize hot regions.

2.4.3 Parallelizing compilers

Parallelizing compilers attempt to automatically discover independent expressions within existing software and generate code capable of exploiting this parallelism. This is a direct response to growing levels of parallelism in conventional CPUs, with multiple cores and the inclusion of vector functional units. Parallelizing compilers exploit parallel processing elements transparently, requiring no effort from programmers to rewrite software or learn new programming paradigms. While most parallelizing compilation passes are implemented in an existing compiler toolchain with access to high-level source code, some work such as [102, 156] attempts to identify and exploit parallelism on raw application binaries.

Programs tend to spend significant parts of their execution times in nested loops, particularly

when processing large data sets. Dense linear algebra, pattern matching, and stencil operations on uniform grids exhibit this property. When loop nests are structured such that data accesses and loop bounds are affine combinations of loop indices and input parameters, they may be modeled as integer *polyhedra*. Each loop iteration is an integer point in a polyhedral space [16] with as many dimensions as nested loops. The polyhedral abstraction enables reasoning about data dependencies between iterations using integer linear programming techniques. These in turn enable numerous compiler-level optimizations that exploit machine characteristics. Common optimizations include loop restructuring, strip mining for vector processing, and coarse-grain partitioning for multithreading. Considerable research into nested loop transformations in a polyhedral framework have made the polyhedral model sufficiently practical that it has begun to appear in modern optimizing compilers.

Polyhedral frameworks are limited to affine loop nests, yet less structured parallel regions are common outside of scientific and engineering workloads. Ryoo et al. [131] identify several analysis techniques for identifying parallelizable regions beyond parallel loopp structures. Bridges [18] describes methods for automatic thread extraction and shows that scalable parallelism can be achieved with extensions to existing sequential programming languages that define a range of legal results. By making very limited source code changes to the SPEC INT 2000 benchmark, existing parallelizing analyses and transformations achieved a speedup of over 4x for a quad-core processor. This result indicates hardware parallelism is under utilized due to the lack of suitable interfaces between software and optimizing compilers for expressing thread-level parallelism. They also propose source annotations facilitating commutivity analysis [6, 130] to identify commutative and thus parallelizable procedures.

Thread-level speculation (TLS) [140] executes iterations of loops as if they are independent, buffering speculatively written state until the program resolves uncertainty about data dependencies between threads. To be efficient, TLS requires hardware functionality to manage and rollback speculative state as well as software to manage the interface to hardware TLS support. These

identify the beginning and end of speculative regions, issue commits when an execution may be safely made non-speculative, and recover in the event of misspeculation. TLS is improved with whole-program analysis [148] in which the compiler's scope for speculation is increased.

2.4.4 Compilation for Heterogeneous Systems

Efforts to construct automatically parallelizing compilers have achieved some successes identifying thread-level parallelism but insufficiently address the problem of determining how these threads should be compiled and executed on heterogeneous processors and systems. Compiling software for heterogeneous architectures is complicated by vastly distinct execution models and programming models needed to target them. A considerable body of ongoing work examines the feasibility of compiling applications for multicore CPUs, GPUs, and other accelerators.

OpenMP [1] extends serial programming model by extending the C++ and Fortran programming langauges with explicit directives for parallelism. An application executes in a single thread until an annotated parallel region is encountered which is then executed across multiple worker threads. These threads are joined at the conclusion of the parallel section. These pragmas are typically applied to loops without loop-caried dependencies enabling the compiler and runtime to partition the iteration space across several threads. OpenMP also includes directives for specifying levels of data sharing, reduction operators, and thread creation. OpenMP presents a non-intrusive approach to exploiting parallelism without significant departures from accepted programming models, but achieving high performance requires architecture-aware optimization decisions that cannot be statically tuned for all classes of hardware.

NVIDIA's Compute Unified Device Architecture (CUDA) [124] is a programming language and toolchain compiling CUDA, a data-parallel extension of C++, for execution on NVIDIA GPUs. OpenCL [66], first defined by Apple and then standardized by Khronos Group's OpenCL Consortium, specifies similar semantics for a kernel-oriented programming model targeting GPUs and multicore CPUs. MCUDA [142] presents a source-to-source transformation for programming

multicore CPUs with CUDA. Compilers from OpenCL to the Cell Broadband Engine [106] and to x86 CPUs via Intel's and AMD's [68] compilers cast OpenCL as a vendor-neutral execution model and language spanning all mainstream heterogeneous processors currently available. With the exception of Cell SPE support, many of these features had been implemented and evaluated in GPU Ocelot before they have appeared in the literature.

Intel's Array Building Blocks (ArBB) [120] addresses growing levels of heterogeneity by presenting a dynamic compilation framework for implementing data-parallel applications. Compute kernels are written using language primitives that perform standard vector processing operations such as map, reduction, and scan. The program is compiled using an off-the-shelf C++ compilation environment, but its execution does not immediately perform the desired computation. Rather, the developer's program is executed in *capture* mode, and the relationship between data structures and operations is used to construct a data-flow graph of the kernel. This becomes the input to a JIT compilation path within ArBB which distributes data structures across worker threads, performs native code generation of the compute kernel, caches the resulting binary, and executes the kernel across participating cores. Intel envisions this approach targeting multiple backend architectures such as symmetric multiprocessors and accelerators.

Other works attempts to compile existing programming models to GPUs. [109] proposes a method for compiling OpenMP applications to GPUs. Lee et al. [108] define a data-parallel subset of the Haskell programming language and uses this to construct CUDA kernels. Copperhead [24] employes a similar paradigm, starting with a subset of the Python programming language. These techniques indicate data-parallel computations may be efficiently compiled for a wide range of processor architectures. Moreover, each of these toolchains emits CUDA as an intermediate program representation that is then statically compiled for execution on GPUs. These efforts demonstrate that existing programming models may be compiled for data-parallel architectures and that the CUDA execution model is a suitable intermediate representation.

A relatively recent body of research focuses on optimization techniques for GPU programs

characterized by predictable control-flow and data movement patterns. Volkov [152] describes a programming technique for eliminating data-transfer overheads in compute-bound kernels through effective tiling and use of the register file. This approach may be generalized with register packing [44] and used to optimize reduction trees in shared memory. G-Streamline [159] is an online runtime solution for detecting and avoiding control-flow and data-access irregularities by remapping thread IDs onto threads. This yields an outcome similar to dynamic warp formation [59] which uses hardware mechanisms to dynamically reform warps to maximize SIMD utilization while avoiding data-transfers within the register file during the remapping. Han et al [71] describe a compiler optimization in which thread-invariant code is hoist out of divergent regions, reducing their size and avoiding redundant serialized execution by diverged warps.

2.5 Conclusion

Dynamic compilation creates flexibility in the interface between software and hardware. The previous model of shipping statically compiled binaries ignores the diversity in hardware and software platforms on which these programs are to execute, missing opportunities to improve performance. CPU designers have made tremendous advances in restructuring instruction streams dynamically without modifying interfaces to software, but there are limits to the types of optimizations hardware is capable of discovering. Dynamic compilation is poised to perform optimizations across a wider view of the application and its environment considering machine parameters, characteristics of input data sets, and interactions with the operating system and other programs. Dynamic optimizing compilers and virtual machines have enabled greater hardware diversity, bridging instruction sets to achieve portability and power efficiency targets.

Parallelizing and heterogeneous compilers have taken steps toward enabling parallel processor architectures. These target computationally intensive loop nests, restructuring them to express

greater levels of parallelism. Loop restructuring based on polyhedral methods offer a rich mathematical framework for code motion and vectorization but depend on strict forms of program structure. Commutatitivity analysis identifies parallel procedural dependencies which may be executed concurrently. Thread-level speculation require less program structure but incur heavy overheads and depend on additional interactions between hardware and software. Together, efforts to improve automatic parallelization of programs have produced tremendously valuable compiler analysis for inferring data dependencies within compute-intensive program regions. Applying these techniques to explicitly data-parallel programs remains an unexplored area of research.

Heterogeneous systems are diverse, by definition, and benefit from decoupling programming models and languages from particular architectures. Moreover, heterogenity benefits applications whose subtasks favor particular types of processors. A dynamic compilation and execution environment with feedback-directed optimization is poised to recompile particular parts of a program for a particular processor. This motivates online profiling and dynamic compilation to detect which parts of an application are best suited for available processors and whether performing that computation on the processor is likely to result in a performance or efficiency gain.

CHAPTER III

CHARACTERISTICS OF HETEROGENEOUS WORKLOADS

Heterogeneous computing assumes different computations are better suited to specialized architectures. Consequently, characterization studies are needed to understand how best to utilize processor architectures and what programmers are able to accomplish without applying heroic levels of manual optimization effort. This characterization study begins with a focus on library-based approaches to leverage heterogeneous processors as well as an effort to parallelize and optimize a computationally intensive matrix computation. Beyond insights gleaned from this development effort and evaluation, this workload characterization defines several metrics and introduces the GPU Ocelot dynamic compilation and simulation framework for executing candidate workloads and analyzing instruction and memory traces.

3.1 Library-based Approach to Heterogeneous Computing

The utility of off-the-shelf libraries for common operations is compelling, as singular efforts to develop and optimize important algorithms are reused in numerous applications. Novel architectures, particularly those that are difficult to program using standard development tools and programming models, may be exposed to application developers as calls to library functions, greatly simplifying their usage. To avoid platform dependencies, libraries provide a convenient and constrained interface that alternate implementations may satisfy for systems lacking specialized accelerators, thus achieving portability.

However, libraries present several constraints in flexibility and performance. Applications may only execute computations defined in the library which may be prohibitive if the library is not well-designed or if it fails to consider or accommodate requirements that are unusual or beyond its

scope. Secondly, libraries constrain the way computations are composed and occasionally force inefficient program behavior as a consequence of how the library is developed and exposed to the machine. For example, a static library cannot easily replace a sequence of element-wise vector operators with a single function and is forced to stream data through main memory for each operation.

This case study evaluates the methodology of library-based approaches to utilizing GPUs via implementing the Vector Image Signal Processing Library [135] API, a standardized application programming interface for dense linear algebra, an application domain particularly suited to GPU computing. VSIPL defines a user-managed consistency model for defining mathematical primitives and includes interfaces for executing dense linear algebra computations. This case study also includes an implementation of QR decomposition for GPUs. This is a matrix factorization that has been historically challenging to parallelize on a bulk-synchronous processor despite its ubiquity in signal processing applications. As an exercise to understand critical algorithm design choices and sensitive optimizations for non-trivial algorithms, this work began with the implementation and optimization of dense linear algebra computatoins for GPUs.

Commodity GPUs are inexpensive resources for delivering very high computing throughput for certain classes of applications. GPUs are sold primarily as an integrated component in display adapters for desktop personal computers. High-throughput GPUs are primarily aimed for the video game market. The primary application of GPUs has a very large degree of potential parallelism at the most computationally intensive step: applying final shading effects to each pixel in a polygon as it is rendered into the display buffer. This fact has allowed GPU vendors to exploit microarchitecture parallelism for increased performance without constraint by the application and without requiring much architectural infrastructure to facilitate parallel execution. The volume of the GPU market provides tremendous competitive pressure to improve performance and keep prices low over successive product generations.

3.1.1 Case Study: QR Decomposition on GPUs

Fully exploiting the peak performance capacity of GPUs has remained a challenge. Algorithms with very high arithmetic intensity, very little need to synchronize between execution paths, and very few scatter operations typically perform well on GPUs without the need for careful optimization. However, many computing tasks do not satisfy these idealized constraints. This case study attempts to accelerate a fundamental matrix computation with GPU computing. First, avaialble algorithms are evaluated and bottlenecks analytically determined. Then, the algorithm is modified to restructure bulk matrix updates as matrix multiplies of large panels rather than single rows or columns. Finally, the resulting algorithm is implemented with efficient and hand-optimized CUDA kernels. This effort reveals several insights into developing efficient implementations of standard problems for utilizing massively parallel processors and gives rise to several metrics related to efficiency and performance. This section summarizes work originally published in [94].

Background. QR decomposition factors an m-by-n matrix A into the product $A = QR$, where Q is an m-by-m unitary matrix and R is an m-by-n upper triangular matrix. QR decomposition is frequently utilized for solving linear systems, conditioning data upstream of additional processing, and in reducing problem sizes of rectangular matrices for subsequent compositions such as Singular Value Decomposition.

This case study examines two-sided factorization algorithms: (1) Givens rotations and (2) Householder reflections [64]. Both of these apply a set of orthogonal transformations to the input matrix to bring it into upper triangular form. By concatenating these orthogonal transforms, the matrix Q is computed, while preserving the invariants $A = QR$ and $Q^H Q = I$. Each of these methods has favorable numerical properties and offers some parallelism. The suitability of each algorithm for GPU implementation depends on memory access patterns, the frequency of inter-processor synchronization, and the scalability of parallelism within the algorithm.

Both the Givens rotations and Householder reflections algorithms for QR decomposition each

exhibit a high degree of parallelism, but have low arithmetic intensity and require frequent communication between processing elements after small numbers of arithmetic operations. As a result, attempts to exploit GPUs to accelerate QR decomposition have been moderately successful achieving 4x speedup [92] over a reference implementation distributed with Lincoln' Laboratory's HPEC Challenge [72]. In this case study, we base speedup numbers on the highly optimized Intel Math Kernel Library which is among the fastest QR implementations available for multicore CPUs.

In the Givens method of QR, a sequence of rotations applied to the input matrix A place zeros in the trapezoidal submatrix below the main diagonal. Each rotation $G(\theta)$ is a Givens rotation, a unitary matrix chosen such that

$$G(\theta) \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} c & s \\ -s^* & c \end{bmatrix} \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

The algorithm begins by choosing elements from the bottom two rows in A in the left-most column. These determine the Givens rotation $G_{m,1}(\theta)$ which may then be used to multiply in-place the submatrix of A formed by the bottom two rows. Then, $G_{m,1}(\theta)^H$ may be used to multiply a pair of columns $Q(:, m-1:m)$ in place. The multiplication in A has the desired effect of overwriting the bottom left element with 0 while preserving the invariants in the algorithm. Next, the algorithm proceeds up one row in A and repeats, computing $G_{m-1,1}(\theta)$ and applying it to $A(m-2:m-1,:)$ and $Q(:, m-2:m-1)$. This proceeds throughout the entire column stopping at the main diagonal at which point all of the left column in A except the first element is overwritten with zeros. The algorithm moves right one column and repeats from the bottom row, again stopping at the main diagonal. When all columns have been visited, A is in upper triangular form, and the QR decomposition is complete.

$G(\theta)$ may be computed from f and g without explicitly computing θ or evaluating any trigonometic functions [13]. Moreover, Sameh and Kuck [132] demonstrate a pattern in which Givens rotations may be computed in parallel. This method is adaptable to streaming architectures and systolic

arrays such as MIT RAW [78]. In general, this approach achieves good performance on MIMD architectures that support low-latency communication and synchronization. GPUs, on the other hand, do not offer on-chip mechanisms for synchronizing multiple Streaming Multiprocessors (SMs) and require either invoking a sequence of kernels with implicit synchronization barriers between each invocation or assuming a consistency model for global memory and implementing barriers in the shared global address space. Performance results for Givens QR implemented on CUDA-compatible GPUs have been presented for the HPEC Challenge benchmarks [92]. However, attempts to develop a high performance implementation of this algorithm in CUDA were met with limited success and do not scale well to arbitrarily large matrix sizes.

Householder QR computes the upper triangular matrix R from an m-by-n matrix A by applying a sequence of *Householder reflections* to A in place [64]. A Householder reflection is an orthogonal transform of the form

$$P = I - \frac{2}{v^H v} v v^H$$

where v is a *Householder vector*. v may be chosen from a vector x such that $Px = e^{j\theta}||x||e_1$, where P is unitary and e_1 is a column vector with 1 in the first element and 0 in all other elements. Part of column k of a matrix A denoted x_k is chosen such that the first element $x_k(1)$ is on the diagonal and the rest of x_k occupies the lower part of A. A Householder reflection P_k computed from x_k may be applied to A in place overwriting column x_k with $P_k x_k$ and updating other columns. We see that $P_k x_k$ is nonzero only in the first element, and all elements below the main diagonal of $P_k A$ are now 0.

Figure 4 illustrates how a sequence of Householder transforms may be chosen from columns of A to bring it into triangular form. In the figure, the dashed rectangle highlights the vector x_k from which a Householder vector v_k is computed. v_k is used to construct the unitary matrix

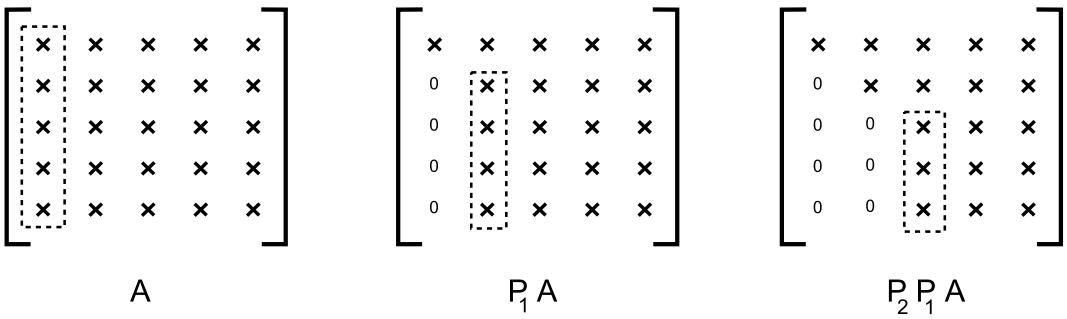


Figure 4: Triangularizing A with Householder reflections.

$P = I_{m-k+1} - \beta v_k v_k^H$. By overwriting A with $P_k A$, zeros in column k are placed below the main diagonal. Moving to the right one column and down one row, the process repeats until A is triangularized. Because P_k is unitary, this sequence may be applied while maintaining the invariant that $A = (P_1^H P_2^H \cdots P_{n-1}^H)(P_{n-1} \cdots P_2 P_1 A)$. We see this is the QR decomposition with

$$\begin{aligned} Q &= (P_1^H P_2^H \cdots P_{n-1}^H) \\ R &= (P_{n-1} \cdots P_2 P_1 A) \end{aligned}$$

Applying a Householder reflection does not require a general matrix-matrix product, for

$$\begin{aligned} PA &= (I - \beta v v^H) A \\ &= A - v w^H \end{aligned}$$

where $w = \beta A^H v$. This gives rise to the following QR algorithm. The function **house**(x) returns the Householder vector $v = x - e^{j\theta}||x||e_1$, where $x(1) = re^{j\theta}$, and $\beta = \frac{2}{v^H v}$.

Require: $Q^H Q = I$
1: $Q \leftarrow I$
2: **for** $k = 1$ to n **do**
3: $[v, \beta]$ = **house**$(A(k : m,\ k))$
4: $A(k : m, k : n) = A(k : m, k : n) - \beta v v^H A(k : m, k : n)$
5: $Q(1 : m, k : m) = Q(1 : m, k : m) - \beta Q(1 : m, k : m) v v^H$
6: **end for**

Algorithm 1: Compute the QR factorization of A via Householder reflections [64].

3.1.2 Blocked Householder QR

Algorithm 1 is conceptually simple and dominated by matrix-vector multiplies. However, the amount of computation per memory element fetched from global memory is quite low. To improve performance, an algorithm in which several Householder transforms may be applied in a single operation was sought. Bischof and Van Loan [14] explain how the Householder QR algorithm may be generalized to represent multiple Householder transforms as a single transformation matrix. Rather than apply Householder reflections as rank-1 updates to the identity matrix,

$$\begin{aligned} P &= P_1 P_2 \cdots P_r \\ &= (I - \beta_1 v_1 v_1^H)(I - \beta_2 v_2 v_2^H) \cdots (I - \beta_r v_r v_r^H) \end{aligned}$$

the m-by-r matrices W and Y are computed such that

$$\begin{aligned} P_{wy} &= P_1 P_2 \cdots P_r \\ &= I + W Y^H \end{aligned}$$

The above operation is rich in matrix-matrix products and can be expected to achieve high performance on GPUs given sufficiently large problem sizes. Moreover, $P_{wy}A$ may be computed as $P_{wy}A = A + W(Y^H A)$ requiring $5mnr$ floating-point operations (FLOPs) if the operations are

performed in the order indicated by the parentheses. This is fewer FLOPs than multiplying the m-by-n matrix A by the m-by-m matrix P_{wy}. Block size may be chosen based on the shared memory capacity of the target architecture and the warp size that leads to maximum performance for matrix multiply. The formation of W may be performed from a block of Householder vectors stored in the lower trapezoidal part of V and from the vector B containing corresponding β_h. The algorithm to form W and Y from r Householder vectors is as follows.

1: $Y = V(1 : end, 1)$
2: $W = -B(1) \cdot V(1 : end, 1)$
3: **for** $j = 2$ to r **do**
4: $\quad v = V(:, j)$
5: $\quad z = -B(j) \cdot v - B(j) \cdot WY^H \; v$
6: $\quad W = [W \; z]$
7: $\quad Y = [Y \; v]$
8: **end for**

Algorithm 2: Computation of W and Y from V and B [14]

Algorithm 2 may be modified by partitioning the columns of the input matrix A into blocks of r columns. For block k, r Householder reflections are computed and applied to triangularize the columns of that block as in the original algorithm. Rather than apply a single Householder transform to the columns of the remaining blocks as before, we instead compute W_k and Y_k as in Algorithm 2 and then apply $P_{wy} = I + W_k Y_k^H$ to the remaining blocks of A and to all of Q. The matrix-vector products are performed on matrices of size $(m - kr) \times r$, and the updates to the remaining blocks and to Q are accomplished by matrix-matrix products. r should be chosen to minimize total runtime on the target architecture. For $r = n$, the blocked Householder algorithm degenerates into Algorithm 1. The entire procedure is specified in Algorithm 3. Workloads in units of real floating point operations are expressed for each phase of the above algorithm in Table 2 for problem sizes of interest.

Require: $A \in C^{m \times n}$, $Q^H Q = I$

$Q \leftarrow I$
for $k = 1$ to n/r **do**
  $s = (k-1) \cdot r + 1$
  for $j = 1$ to r **do**
    $u = s + j - 1$
    $[v, \beta]$ = **house**$(A(u : m, u))$
    $A(u : m,\ u : s + r - 1) = A(u : m,\ u : s + r - 1) - \beta v v^H A(u : m,\ u : s + r - 1)$
    $V(:, j) = [\text{zeros}(j - 1, 1);\ v]$
    $B(j) = \beta$
  end for
  $Y = V(1 : end, 1)$
  $W = -B(1) \cdot V(1 : end, 1)$
  for $j = 2$ to r **do**
    $v = V(:,\ j)$
    $z = -B(j) \cdot v - B(j) \cdot W Y^H v$
    $W = [W\ z]$
    $Y = [Y\ v]$
  end for
  $A(s : m, s + r : n) = A(s : m, s + r : n) + Y W^H A(s : m, s + r : n)$
  $Q(1 : m, s : m) = Q(1 : m, s : m) + Q(1 : m, s : m) W Y^H$
end for

Algorithm 3: Block Householder QR

Table 2: Workload for real-valued blocked Householder QR in GFLOP.

Dimension	**house**$(A(:, u))$	$A = P \cdot A$	WY	$A = P_{wy}^H \cdot A$	$Q = Q \cdot P_{wy}$	Total
512×256	0.000196	0.00463	0.00656	0.0518	0.211	0.275
1024×512	0.000786	0.0184	0.0257	0.433	1.66	2.14
1536×768	0.00177	0.0413	0.0571	1.48	5.55	7.14
2048×1024	0.00314	0.0734	0.102	3.54	13.1	16.8
2560×1280	0.00491	0.115	0.158	6.94	25.6	32.8
3072×1536	0.00708	0.165	0.228	12.0	44.1	56.5
3584×1792	0.00963	0.224	0.310	19.1	70.0	89.7
4096×2048	0.0126	0.293	0.404	28.6	104	134
4608×2304	0.0159	0.371	0.511	40.7	149	190
5120×2560	0.0197	0.458	0.631	55.9	204	261
6656×3328	0.0332	0.773	1.65	123	447	572
8192×4096	0.0503	1.17	25.0	230	833	1090

3.1.3 Optimizing Dense Linear Algebra Kernels

Algorithm 3 was first implemented in C++ with the CUBLAS library distributed with CUDA 2.0 to obtain baseline performance results. Each function call into CUBLAS was instrumented with performance counters to measure the fraction of total runtime spent in each operation. The impact of timing instrumentation on total runtime is small compared to total runtime. The instrumented algorithm was executed five times, and the timing profile across all runs was averaged. Functions dominating runtime were then selected for custom implementations as one or more CUDA kernels.

The CUDA execution model [124] specifies a collection of multiprocessors that share a global address space. Each multiprocessor is composed of 8 datapaths that execute a "warp" of 32 threads in SIMD fashion. When the threads of a warp stall due to reaching a synchronization barrier or memory operation, another warp on the same processor is scheduled for execution. Kernels are designed with several concurrent warps executing on each multiprocessor such that the GPU performs computation while memory transfers complete.

In this implementation, where possible, operations requiring several CUBLAS function calls were combined into single kernels. For example, the norm computation dominates the runtime of the **house**() function. Because each block of A is transformed exclusively by reflections, the norms of the columns of each block are invariant and may be computed in parallel by a single kernel invocation before the block is triangularized. This avoids the overhead of calling shorter kernels many times and utilizes all multiprocessors, with each multiprocessor performing a reduction operation corresponding to a particular column.

The datapaths of a multiprocessor access a large register file partitioned among the many threads and a 16 kB scratchpad known as "shared memory." Shared memory is addressable by load and store instructions and is striped across 16 ports. Each port is 32 bits wide, and if every thread of a "half-warp" accesses a different port, the load or store instruction will not stall. Selecting access patterns to shared memory that avoid port conflicts reduces stall rates and maximizes

throughput. Typically, this requires skewed access to two-dimensional arrays in shared memory among threads of the same warp. Storing blocks of frequently used data in registers reduces pressure on shared memory and avoids additional instructions to required load values into registers before they may be used as operands.

Algorithm 3 was expected to be bottlenecked by matrix-vector products within the main loop of the algorithm. The CUBLAS prototype makes several calls to the CUBLAS function `cublasSgemv()` to compute the product $\beta A'v$ and the product βAv. To improve performance, these were implemented with custom CUDA kernels according to the architecture characteristics discussed in this section. Shared memory is used only to store part of the vector v. Columns of the matrix A are streamed into the register file with coalesced read operations where they are used to multiply corresponding elements of the V vector. The CUDA compiler is capable of automatically unrolling loops with constant cycle counts [[124] §4.2.5.2]. All threads attempt read access to the same address in shared memory, so no bank conflict occurs when loading `V_shared[j]`. To avoid potentially expensive thread divergence, guard conditionals are excluded and matrix dimensions are assumed to be multiples of 64. The following kernel exhibits an average 2.5x speedup over the CUBLAS 2.0 [122] function `cublasSgemv('n',..)` for problem sizes of interest. As described in [153], a relatively small block size of 64×1 threads was selected to reduce overheads associated with loop index and address calculations. This departs from the CUDA Programming Guide which recommends a large number of threads to maximize occupancy [[124] §5.2].

Performance of an m-by-m matrix multiplying an m element vector is illustrated in Figure 5. The target GPUs for this benchmark are the GeForce GTX 280 and GeForce 9800 GX2. Additionally, a theoretical upper bound for floating-point performance in GFLOP/s suggested by global memory bandwidth is plotted for each architecture. For the GeForce GTX280, theoretical bandwidth is 141 GB/s; for the GeForce 9800GX2, theoretical peak bandwidth is 64 GB/s. The number of FLOPs for the `Sgemv` operation is computed as $2mn + 2m$, and total data transferred assuming no redundancy is $4mn + 4n$ bytes. This performance bound ignores kernel launch overheads

```
/*
    Computes W = beta A V + alpha W
*/
__global__ void gtSgemv(
    float alpha, float beta, float *A,
    float *V, int M, int N, float *W) {

    int row = blockIdx.x * 64 + threadIdx.x;
    __shared__ float V_shared[64];
    float w = alpha * W[row];

    A += row;
    V += threadIdx.x;

    for (int k = 0; k < N; k += 64) {
        V_shared[threadIdx.x] = *V;
        V += 64;
        __syncthreads();
        for (int j = 0; j < 64; j++) {
            w += *A * V_shared[j];
            A += M;
        }
        __syncthreads();
    }
    W[row] = beta * w;
}
```

Listing 3.1: Matrix-vector product

which are deemed to be negligible for large problems. As illustrated, the custom `gtSgemv` kernel achieves a maximum of 69 GFLOP/s or 97% of theoretical performance when executed on the GeForce GTX280 for large matrices. Examining the CUBLAS source, we see that the access patterns of `cublasSegmv('n', ..)` are similar to the kernel in Listing 1, but the kernel body includes a significant number of integer operations and shared memory loads and stores, all of which reduce floating point intensity.

Similarly, a kernel computing a matrix-vector product was written to compute $\beta A^H v$. Together, these functions demand the majority of runtime during the triangularization phase and the formation of the WY representation of a set of Householder reflections. All custom kernels access submatrices and vectors beginning on rows that are multiples of 16 to ensure memory transfers are

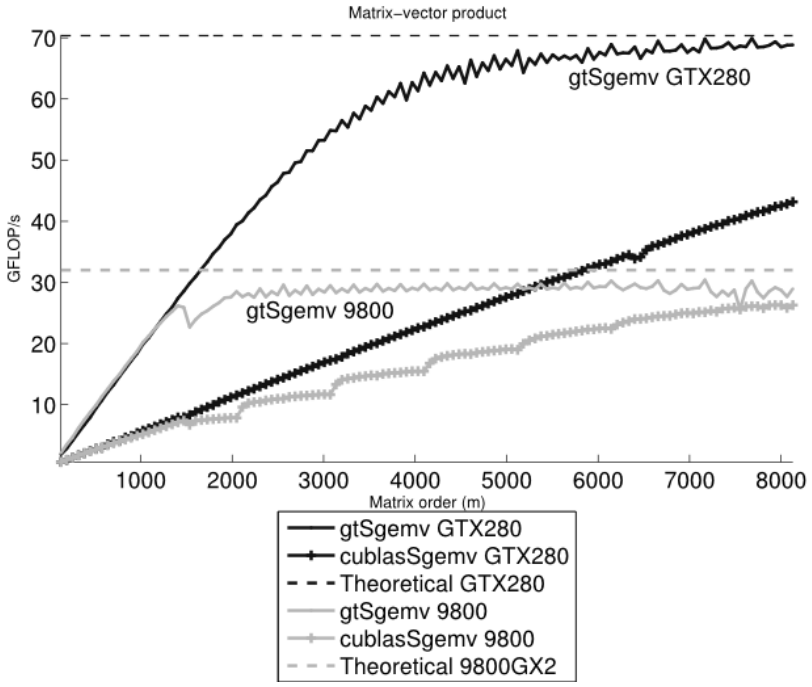


Figure 5: Performance of matrix-vector product for matrices of dimension m

aligned to 64 byte boundaries and complete in as few tranfer operations as possible. The additional elements are overwritten with zeros once they are loaded into registers. CUBLAS matrix-matrix product functions are called to compute the updates for A and Q, lines 19 and 20 in Algorithm 3. Matrix product operations in CUBLAS have been measured to obtain over 400 GFLOP/s sustained and approach peak multiply-add performance for the architectures under test [153].

The performance of this implementation of QR decomposition was measured by computing the QR factorization of real-valued rectangular matrices with twice as many rows as columns corresponding to typical overdetermined least squares problems. The input matrix A was initialized with random values from -1 to 1 below the main diagonal, 0 written above the main diagonal, and 1s along the diagonal to ensure full rank. Then, randomly selected Givens rotations were applied to A to conceal all apparent structure while preserving rank. Although some applications of QR do not require Q to be explicitly formed, performance results presented here include the computation

of the full m-by-m Q matrix. Error criteria were selected as follows.

$$\begin{aligned} ||QR - A|| &\leq m \cdot 2^{-23}||A|| \\ ||Q^H Q - I|| &\leq m \cdot 2^{-23} \\ ||L|| &\leq m \cdot 2^{-23} \end{aligned}$$

where L is the trapezoidal submatrix below the main diagonal of R. All tests for which performance results are reported satisfy these error criteria. The testbed application was compiled and linked with the CUDA 2.0 toolchain and executed on a Q6600 quad-core Intel Core2 at 2.4 GHz running Windows XP. Additionally, a reference application was implemented with the Intel Math Kernel Library 10.0 and executed on a 64-bit Linux machine with a quad-core Intel Xeon CPU at 2.8 GHz. For both applications, the matrix data type was in single-precision, and all matrices were in column major order. For the GPU test application, all data was resident in device memory before timing measurements were made and no PCI-Express bus traffic was considered. This scenario is typical of a real-world GPU application using CUBLAS or GPU VSIPL [93] in which intermediate results are stored on the device, and only final outputs are copied back to host memory.

The QR procedure was instrumented with CPU-based performance counters to record the number of cycles required by each phase of the algorithm. `cudaThreadSynchronize()` was called after each kernel invocation to ensure the kernel had completed before stopping the cycle counter. No numerical processing was performed on the CPU for the GPU algorithm, and runtime is almost entirely a function of the target GPU's performance. This implementation was tested on two GPU architectures supporting CUDA: NVIDIA GeForce 9800 GX2 and GeForce GTX 280. For each target, the CUDA compiler `nvcc` was invoked with a flag for the maximum possible shader model version supported by the target GPU. The flag `-maxrregcount` was issued to `nvcc` to clamp the register usage to 32 registers per thread to avoid spilling to local memory on the 9800 GX2. The register file for the GTX280 is twice as large, accommodating larger warp

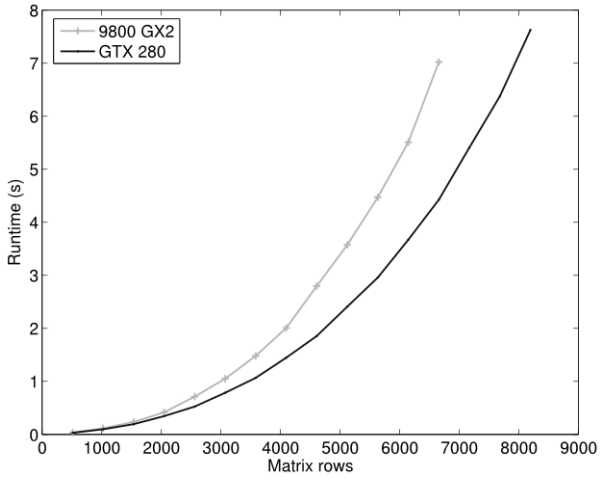


Figure 6: Runtimes of QR decomposition on GPUs.

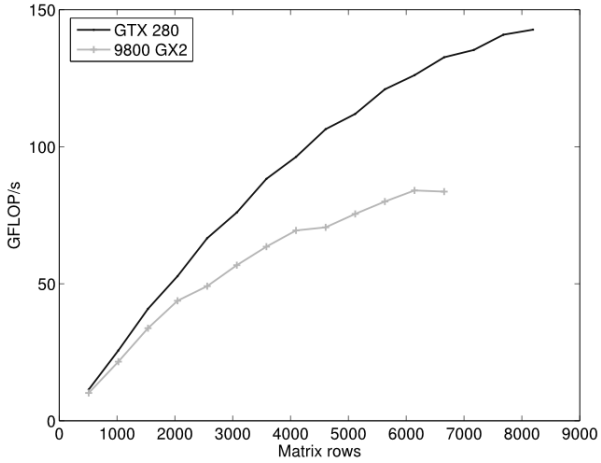


Figure 7: Sustained performance of QR decomposition on GPUs.

sizes without register spilling.

Overall runtime for various matrix sizes is presented in Figure 6, and performance with respect to workload is illustrated in Figure 7. The maximum problem size for the GeForce 9800 GX2 was limited by its 512 MB of global memory per GPU. Figure 8 illustrates the speedup of the GeForce GTX 280 with our QR algorithm over the Intel MKL library's support for QR decomposition implemented by the LAPACK functions `sgeqrf()` and `sormqr()`; MKL was run with 1, 2, and 4 threads.

The GeForce GTX280 achieves 143 GFLOP/s sustained performance with our QR implementation. This constitutes the highest performance of a single processor for real QR decomposition we are aware of. As predicted, triangularizing a block of A and forming the WY representation of the Householder reflections requires much more runtime than applying the blocked Householder reflections to the rest of A. Applying the reflections to Q requires a significant fraction of runtime, as Q is full with m rows and m columns. Moreover, while the submatrix of A to which P_{wy} is applied grows smaller as the algorithm progresses, the number of rows in Q to which P_{wy} is applied remains constant. As Figure 9 illustrates, the large workload in transforming Q achieves the highest performance, as it consists of large matrix-matrix products. Table 2 expresses the distribution

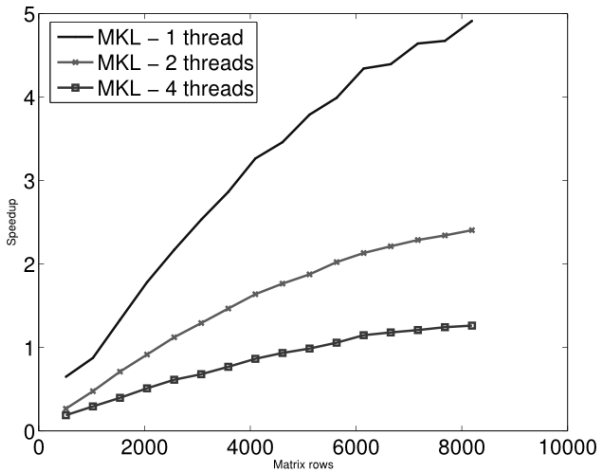


Figure 8: Speedup of GTX280 QR implementation over MKL.

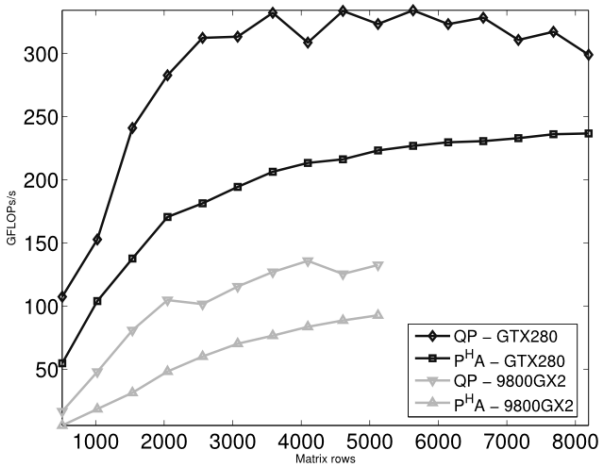


Figure 9: Performance of $A \leftarrow P^H A$ and $Q \leftarrow QP$ for GeForce GTX280 and GeForce 9800.

of runtime across the several phases of our QR implementation for the largest possible problem sizes on both the GeForce 9800 GX2 and the GeForce GTX 280. The fraction of total runtime for each phase did not change appreciably with problem size except for very small matrices.

We have demonstrated an implementation of QR decomposition that runs entirely on the GPU. Restructuring algorithms so they are composed of dense operations on blocks of data takes advantage of high bandwidth to the register file, permitting each multiprocessor to approach theoretical peak performance. Structuring algorithms in terms of operations on blocks of matrices leverages the streaming architecture of CUDA-capable GPUs. Kernels were implemented with detailed knowledge of the underlying GPU architecture and offer performance beyond what is available in CUBLAS 2.0.

Our QR implementation achieves nearly 5x speedup for large matrices over Intel's MKL native QR algorithm. The algorithm selection and implementation details covered here apply to other architectures with deep memory hierarchies and data parallel arithmetic units. In the future, we hope to investigate load balancing between the CPU and the GPU, permitting high-performance libraries such as MKL to take advantage of the CPU's strengths while tasking the GPU with large

block-oriented procedures. Nevertheless, we consider 143 GFLOP/s of sustained performance entirely on the GPU a notable achievement.

This case study has lead to several observations about GPU computing. First, this required carefully choosing an algorithm that is both efficient and scalable. Fine-grain synchronization requirements of Givens rotations ultimately cannot be satisfied by GPUs which lack efficient global barriers. The initial Householder reflections algorithm relies on numerous matrix-vector products and is ultimately limited by off-chip memory performance. While CPUs with large caches may be able to achieve better performance due to large on-chip caches, the results would suffer a severe performance cliff once problem sizes exceed cache capacity. The blocked Householder Reflections algorithm, applying updates in bulk, is much more efficient in terms of computation per byte transferred off chip and showed strong performance scaling across successive generations of GPUs with varying levels of concurrency.

We observe several low-rank updates computed per reflected column may be implemented by computing several matrix-vector products at once. Fusing multiple operations into a single kernel is critical for reducing memory bandwidth demand. However, this requires writing custom compute kernels rather than relying on library-provided matrix and vector operators. Finally, we also note simple kernels are able to achieve near-peak performance for some types of workloads as in the custom matrix-vector product kernel which out-performed CUBLAS's implementation. Attempts to improve upon the performance of CUBLAS, however, were not as successful and required numerous low-level program transformations such as unrolling loops and register blocking.

3.2 Data-Parallel Metrics for Efficiency and Performance

Applications composed of data-parallel kernels targeting heterogeneous systems are relatively new, and consequently the characteristics of viable workloads are not altogether understood from an empirical perspective. The goal of this research effort [95] is to identify representative high-performance GPU applications from several benchmark suites and application repositories, execute

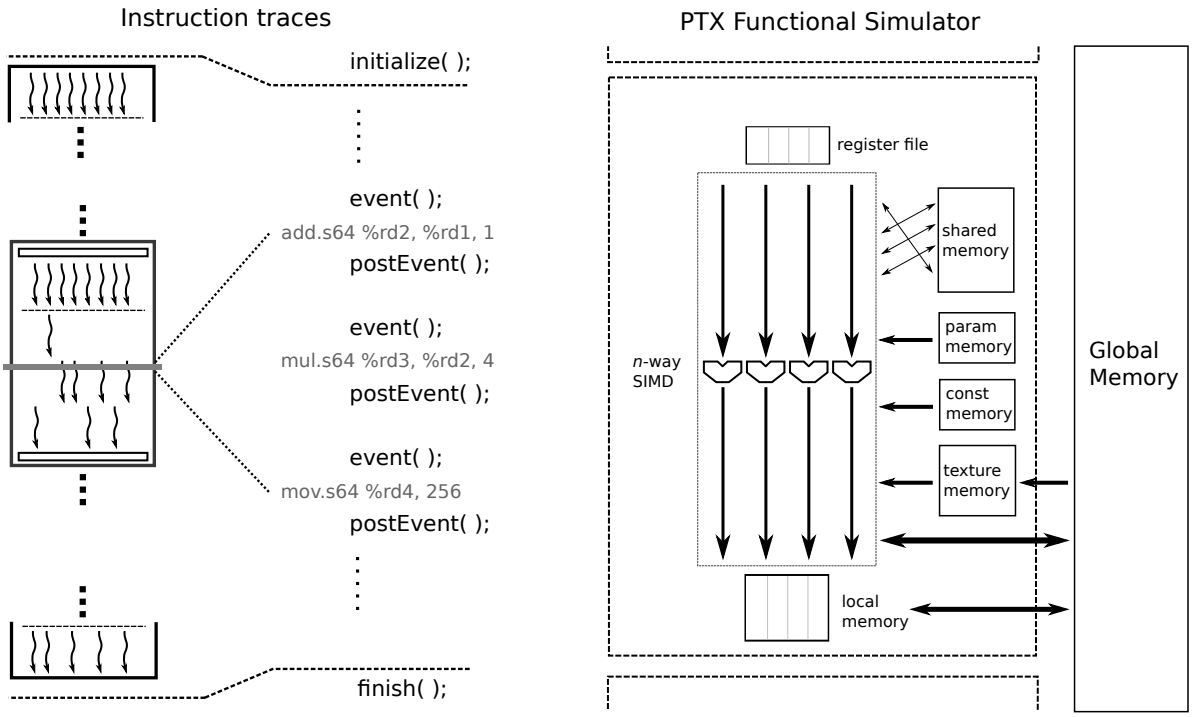


Figure 10: PTX Emulator trace generation facilities with abstract machine model.

these applications on a functional simulator, and then perform analysis on the resulting instruction traces to obtain quantitative application profiles. This work has resulted in defining several key metrics correlated with performance. These are related to dynamic instruction counts, SIMD utilization, memory performance, synchronization, and parallelism scalability.

To perform this analysis, GPU Ocelot's PTX emulator was developed directly implementing the virtual machine defined by PTX and illustrated in Figure 10. As a functional simulator of an abstract GPU, results are not dependent on particular GPU microarchitectures. Rather, they capture behaviors of the *execution model* defined by CUDA and PTX. An interface for gathering instruction traces including thread activity mask, PTX instruction objects, branch targets, and memory addresses was added to Ocelot's emulator backend to enable extensible trace gathering and analysis.

Explorations requiring alternative execution and programming models are faced with the challenge of evaluating optimization techniques on representative workloads. By supporting a sufficiently complete fraction of the CUDA programming language, techniques developed in the course of this research may be evaluated over a large set of existing applications provided from the following external sources.

1. **CUDA Software Development Kit**, a set of programs and utility libraries distributed with CUDA to demonstrate individual capabilities of the language and showcase high-performance available from GPU computing.

2. **Parboil Benchmark Suite** [81] provides a more comprehensive set of applications covering several computational motifs.

3. **Rodinia Benchmark Suite** [26] includes a larger set of applications from medical imaging, bioinformatics, linear algebra and image processing, and unstructured problems such as graph algorithms and cellular automata.

Modeling these workloads on GPU Ocelot's functional simulator yields characteristics of the parallel primitives summarized in Table 3 and the application-level demonstrations summarized in Table 4. From these applications, we derive additional novel metrics.

3.3 Metrics

This chapter defines the following metrics and demonstrates their usefulness in characterizing GPU compute workloads.

3.3.1 Control Flow

Activity Factor. Instructions are executed in SIMD manner across all threads of a CTA. However, control flow instructions may cause threads of a CTA to diverge, as some threads take a branch

Table 3: SDK Building Block Statistics

Building Blocks	Kernels	Average CTA Size	Average CTAs	Instructions	Branches	Branch Depth
Bitonic Sort	1	128	1	1091	176	11
ConvolutionFFT2D	15	85.31	152.6	960129	14681	3
Separable Convolution	4	141.58	304	204480	10112	5
Texture Convolution	2	192	1376	520192	2752	3
Histogram64	2	191.77	119.5	1169535	56969	4
Histogram256	5	189.45	140.8	3309550	293753	5
Matrix Multiply	1	256	40	9760	200	3
Reduction	2	127.89	32.5	41521	4354	3
Scalar Product	1	256	256	148224	16128	6
Scan	3	309.31	1	1455	102	5
Scan Large	10	249.26	8.2	28500	1782	5
Transpose	4	256	4096	802816	24576	2

while others fall through. We use the term *activity factor* to refer to the average fraction of threads that are active at a given time. It is defined as

$$AF = \frac{1}{N} \sum_{i=1}^{N} \frac{active(i)}{CTA(i)} \tag{1}$$

where $active(i)$ is the number of threads active when dynamic instruction i is executed and $CTA(i)$ is the number of threads in the CTA executing instruction i for N total dynamic instructions. For a kernel with no control flow or control flow in which all threads branch or fall through a branch instruction uniformly, thread activity is 100%.

Divergent Branches. The PTX machine model defines a divergent branch as a branch instruction where some threads within the same warp branch while others fall through. In this study, we report the branch divergence of an application as the ratio of divergent branches to total branches. As control flow divergence results in two separate sets of threads whose execution must be serialized, the location of the synchronization point has a profound impact on thread activity and the number of dynamic instructions executed. In [59], Fung et al. show that the earliest (ideal) location of thread reconvergence is the immediate post dominator of the branch instruction - that is, the nearest successor basic block that *must* be executed if the branch instruction is executed regardless of control path taken. We also investigate an alternative method in which divergent threads are not

Table 4: SDK Application Statistics

Applications	Kernels	CTA Size	Average CTAs	Instructions	Branches	Branch Depth
Bicubic Texture	27	256	1024	222208	5120	3
Binomial Options	1	256	4	725280	68160	8
Black-Scholes Options	1	128	480	3735550	94230	4
Box Filter	3	32	16	1273808	17568	4
DCT	9	70.01	2446	1898752	25600	3
Haar wavelets	2	479.99	2.5	1912	84	5
DXT Compression	1	64	64	673676	28800	8
Eigen Values	3	256	4.33	9163154	834084	13
Fast Walsh Transform	11	389.94	36.8	32752	1216	4
Fluids	4	36.79	32.6	151654	3380	5
Image Denoising	8	64	25	4632200	149400	6
Mandelbrot	2	256	40	6136566	614210	26
Mersenne twister	2	128	32	1552704	47072	7
Monte Carlo Options	2	243.54	96	1173898	76512	8
Threaded Monte Carlo	4	243.54	96	1173898	76512	8
Nbody	1	256	4	82784	1064	5
Ocean	4	64	488.25	390786	17061	7
Particles	16	86.79	29.75	277234	26832	16
Quasirandom	2	278.11	128	3219609	391637	8
Recursive Gaussian	2	78.18	516	3436672	41088	8
Sobel Filter	12	153.68	426.66	2157884	101140	6
Volume Render	1	256	1024	2874424	139061	5

reconverged until explicit synchronization barriers are encountered by ignoring compiler inserted reconverge points. We denote the former method *ideal reconvergence* and the latter method *barrier reconvergence*.

Results. There are about 55 million dynamic instructions executed by each CTA, totaled across all applications from the CUDA SDK. As each dynamic instruction is executed in SIMD fashion by multiple threads, this count would be much greater if threads were serialized on a single-threaded architecture. Of these dynamic instructions, roughly 6.5% are branches, and of those branches, 9.5% are divergent. Parboil is comparatively a much larger benchmark with over 4 billion dynamic SIMD instructions across the entire suite. The ratio of branch instructions is slightly higher at 9.01%, and the ratio of divergent branchs to branches is significantly higher at 31.7%.

Moving from ideal reconvergence at the immediate post dominator to reconvergence at the next barrier instruction increases the average number of dynamic instructions significantly by a factor

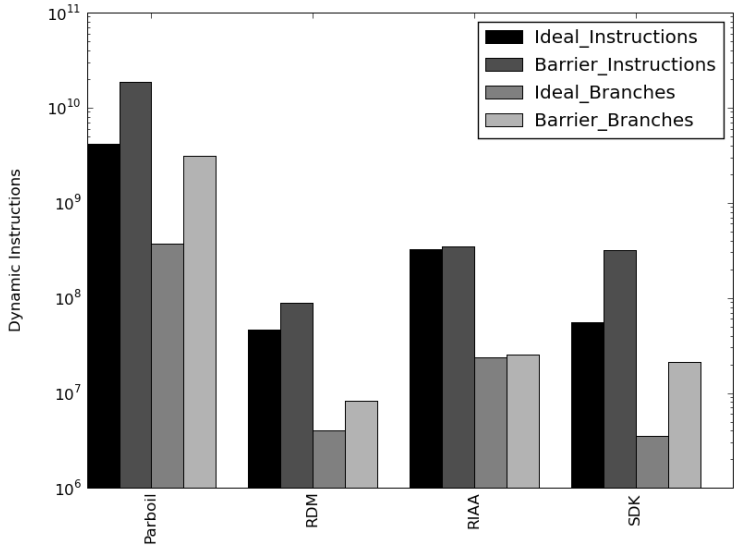


Figure 11: Dynamic instruction and branch counts for ideal and barrier reconvergence.

of 5.72 for the SDK as shown in Figure 11, which presents dynamic instruction counts using ideal reconvergence as well as barrier reconvergence for each workload. Total branch instructions using both reconvergence algorithms is also presented. The RDM and Parboil workloads have a similar response, increasing by a factor of 1.91 and 3.84 respectively. The RIAA application, on the other hand, only increases slightly by 1.06x.

The activity factor is also dramatically impacted as can be seen in Figure 12, dropping from 85.15% to 20.35% across the SDK. When looking at the SDK in detail, some applications like Histogram256 are affected dramatically by the warp convergence mechanism, while others like Bitonic are not affected at all. One explanation is that some applications have very few to no divergent branches and consequently are unaffected by the divergence mechanism. A second explanation is that the code is structured such that the placement of the barrier largely coincides with the behavior of the post dominator scheme. This is the case for RIAA which exhibits substantive control flow divergence but whose performance is minimally impacted by the reconvergence mechanism.

Figure 12 shows that the ratio of divergent branches to total branches decreases by 15x using

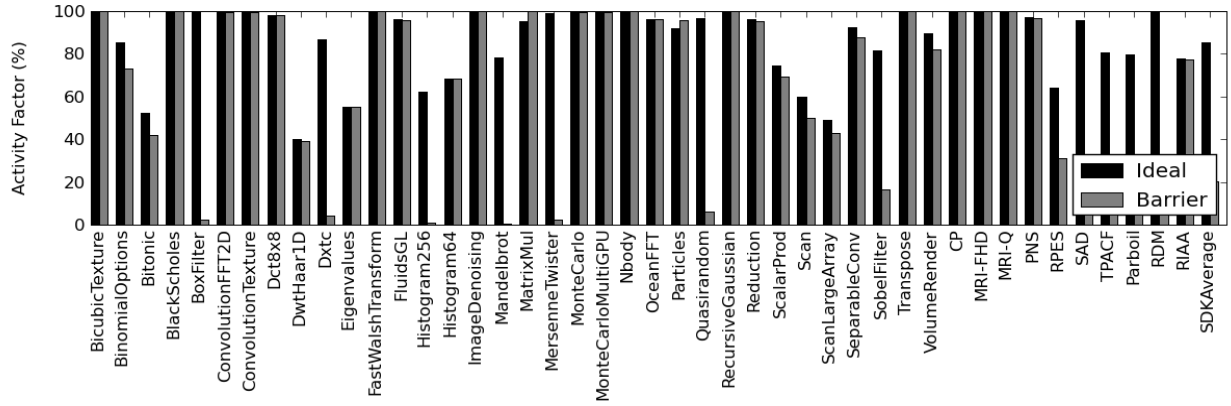


Figure 12: Activity factor for ideal and barrier reconvergence.

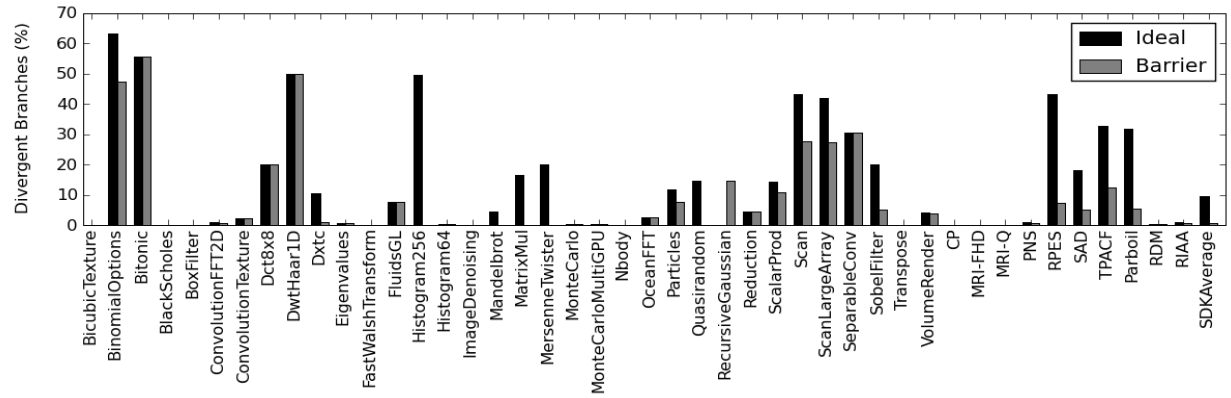


Figure 13: Fraction of divergent branches for two reconvergence mechanisms.

barrier reconvergence in the CUDA SDK Average suggesting that consecutive branches are correlated, and the ideal reconvergence scheme leads to situations where warps are recombined and then immediately split again. If the programmer has intuition as to how long threads will remain divergent, it would be beneficial to allow them to specify the convergence points if the execution cost of splitting and recombining warps is high. However, as long as the hardware cost of splitting and recombining warps is low, it would seem preferable to converge as soon as possible at the immediate post dominator, since this scheme performs at least as well and often significantly better in terms of activity factor and dynamic instructions as the barrier scheme in all applications that we have examined.

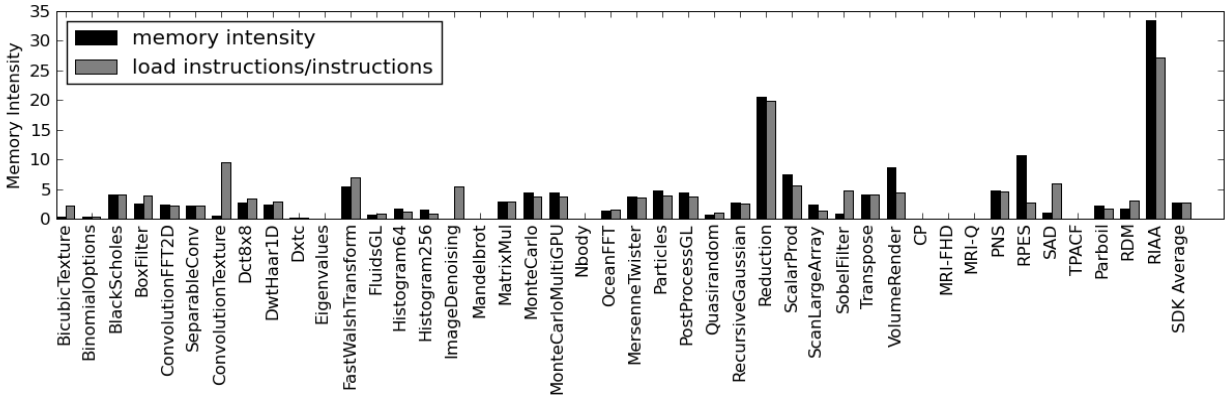


Figure 14: Memory Intensity

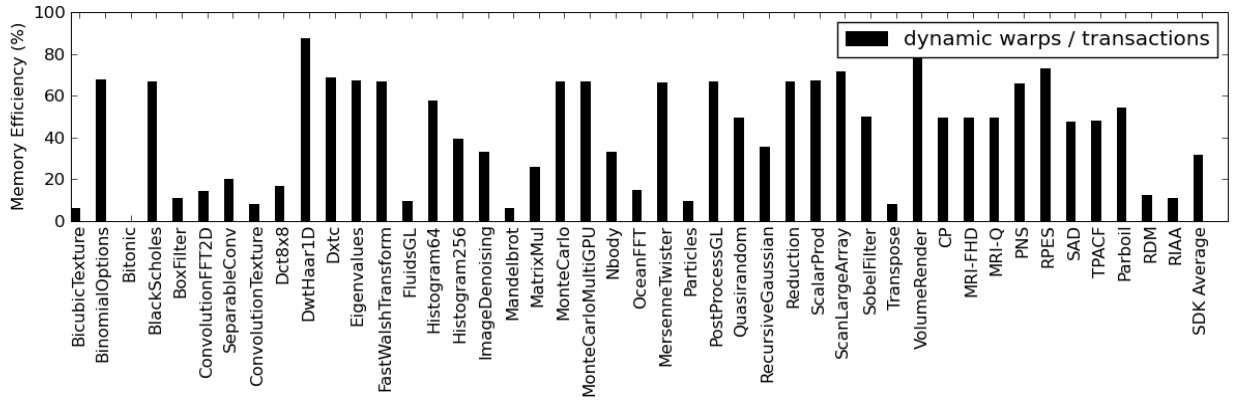


Figure 15: Memory Efficiency

Recommendations. Several applications such as D̂wtHaar1D, *Mandelbrot*, and *RIAA* exhibit control flow behavior that includes handling of special cases that necessarily result in lower thread activity. Similarly, applications such as *Histogram256* exhibit correlated branches. In these cases, grouping threads by such affinity (user or compiler optimizations) would be beneficial from an activity factor perspective. For example, the *Particles* example performs different computations depending on the scene being simulated. Additional support via profiling or prediction coupled with dynamic recompilation or hardware remapping of threads to CTAs (and/or warps) as in [59] would enable efficient use of such SIMD architectures.

3.3.2 Memory Behavior

To characterize the memory demand of PTX programs, we define the metric **memory intensity** (I_M) as the ratio of the total number of operations to global memory (M_i) to the number of dynamic instructions (D_i) multiplied by the activity factor (A_f) as in Equation 2. For the purposes of this metric, texture sampling is counted as a load of four 32-bit words. Memory intensity for all workloads is presented in Figure 14.

$$I_M = A_f \times \frac{\sum_{i=1}^{kernels} M_i}{\sum_{i=1}^{kernels} D_i} \tag{2}$$

PTX defines six address spaces: constant, global, local, param, shared, and texture. Constant, param, and texture memory are read-only and backed by a cache [124]. Global memory is the largest block of memory in the memory hierarchy and also that with the largest latency. The PTX memory model enables multiple threads accessing words in the same 64- or 128-byte segment of global memory in the same load or store instruction to coalesce these accesses into a single transaction. This may result in one or two memory operations depending on the width of the address bus and size of the transaction. Scatter operations in which each thread accesses a word in a unique segment result in one transaction per segment greatly reducing useful memory bandwidth on real-world platforms.

To characterize spatial locality of operations to global memory, the CUDA memory coalescing protocol for compute capability 1.2 was implemented [124]. This protocol groups operations made by a SIMD load or store instruction into transactions that each access contiguous segments of memory. While Ocelot largely ignores the concept of a warp - a set of consecutive threads that are executed on the hardware's SIMD units concurrently - this metric may only be computed assuming a particular warp size. In this case, we choose 32 threads per warp corresponding to NVIDIA's GT200 architecture. Each memory instruction produces at least two transactions, one for each half-warp. Memory efficiency (E_M) is therefore defined in terms of the number of dynamic warps

($W_{i,j}$) executing each global memory dynamic instruction and the number of memory transactions needed to complete these instructions ($T_{i,j}$) for kernel i and CTA j. A dynamic warp is a warp containing at least one active thread.

$$E_M = \sum_{i=1}^{kernels} \sum_{j=1}^{CTAs} \frac{2W_{i,j}}{T_{i,j}} \tag{3}$$

Memory bandwidth utilization approaches theoretical peaks as **memory efficiency** approaches 100%. Figure 16 illustrates how a load of consecutive addresses within the same warp is coalesced into a single request, yet a store to strided addresses results in four requests. As the number of transactions required to satisfy a warp's loads or stores increases due to scattered access patterns, this ratio decreases. Memory efficiency for the CUDA SDK, Parboil, RDM, and RIAA applications are presented in Figure 15.

Results. Applications in the evaluated workloads exhibit low memory intensity. The CUDA SDK average memory intensity is 2.70%, Parboil's is 1.71%, and the RDM application's is 3.02%. This indicates the possibility they are compute bound. Applications with particularly high memory intensity include several regression tests from the CUDA SDK that intentionally stress the memory system as well as the outlying RIAA application in which memory intensity is 27.1%. The CUDA SDK average memory efficiency of 31.0% indicates, on average, three memory transactions are required to satisfy each memory instruction. Among these workloads, applications with the least memory efficiency tend to be those that rely heavily on texture sampling.

Recommendations. Several applications exhibit low memory efficiency, but high activity factor. In other words, even though threads are executing the same instructions, they are not accessing spatially local data. Previous solutions that focus on hardware memory coalescing can only do so much in these cases as threads within a warp that each access different DRAM pages must issue separate transactions. We are exploring the use of memory profiling of threads and automatically grouping them into warps based on the locality of their memory accesses. Such warp formation

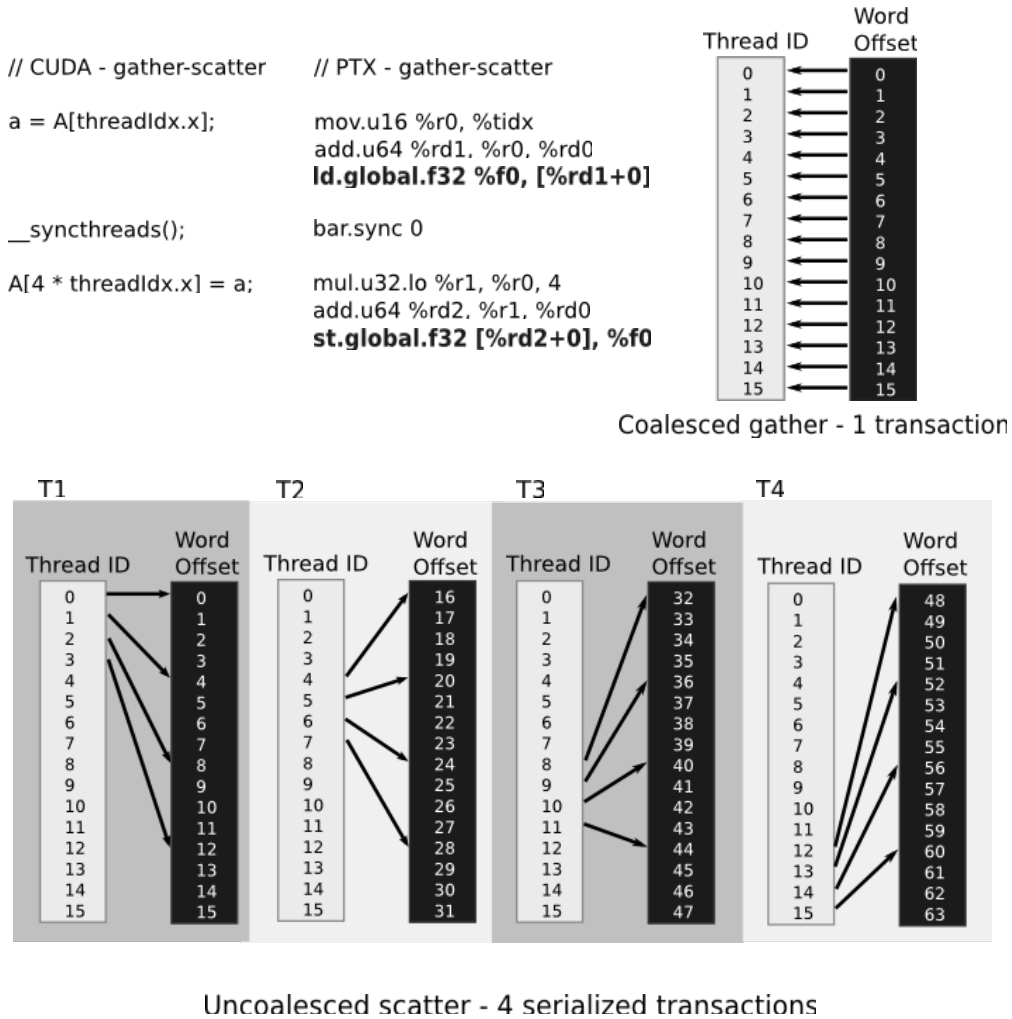


Figure 16: Coalesced gather followed by an uncoalesced scatter illustrating memory efficiency metric.

must be contrasted with the impact on branch divergence behavior.

3.3.3 Data Flow

Data Sharing. Cooperative Thread Arrays have access to a scratchpad memory block known as **shared memory**. Shared memory provides a mechanism through which data may be exchanged among the threads of a CTA. To characterize workloads with producer-consumer behavior among threads of the same CTA, stores to shared memory are tracked to corresponding loads from other threads within the CTA. Each byte of data may be annotated by the ID of the thread that last stored

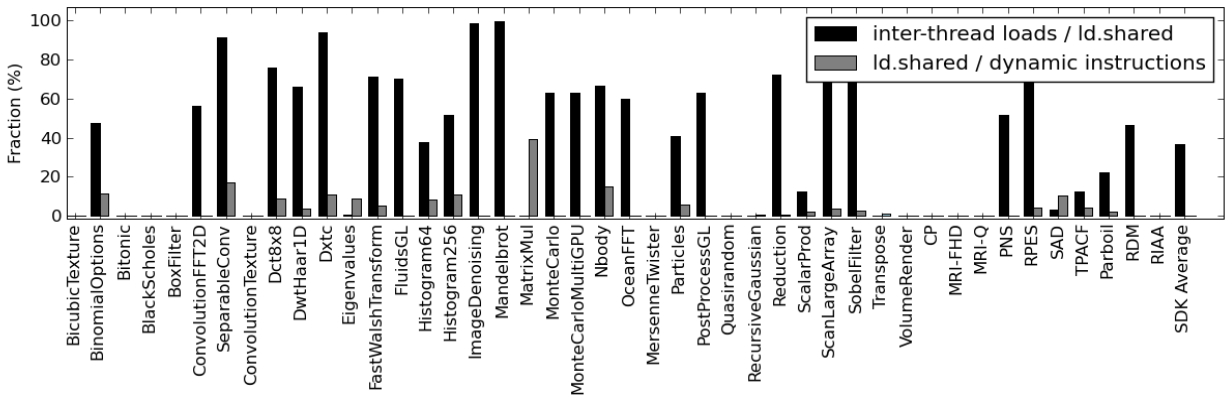


Figure 17: Inter-thread Data Flow within shared memory.

data to that address. Loads from shared memory examine the producing thread ID for that byte, and the load is marked as an inter-thread load if the producing ID is different from the ID of the thread loading the data. The metric we report, **inter-thread data flow**, is the number of inter-thread loads from shared memory relative to the total number of words loaded from shared memory (within the CTA).

To distinguish between true producer-consumer relationships among threads from uses of shared memory as an effective software-managed read-only cache, stores to shared memory do not update the producing thread ID annotation if the register containing the data to store was last written by a load from global memory within the same basic block. This avoids counting uses of shared memory intended to accommodate global memory coalescing and focuses on cases in which the computed results of one thread are consumed by another. This is presented for all workloads in Figure 17. To illustrate the impact of shared loads on the entire kernel, the fraction of shared loads to total dynamic instructions also appears in the figure.

Results. Several applications such as the Monte Carlo simulations (Black-Scholes, Monte-Carlo, and RIAA) do not make use of shared memory at all and exhibit no interdependencies among the threads. This suggests that threads may be arbitrarily grouped (into warps) within CTAs for example to improve the activity factor. Other applications such as matrix multiply use

shared memory to broadcast data streamed in from global memory and could be accommodated by architectures with data caches. Many applications, however, exhibit a high fraction of inter-thread loads. The CUDA SDK averages 36.7% of inter-thread load instructions reading words produced by other threads. The RDM application is dominated by FFTs with over 45% of shared loads constituting inter-thread data dependencies. The Parboil benchmarks share relatively less data between threads (22.1%), but have a significant fraction of shared loads to total instructions (2.3%).

Recommendations. Many applications in this selection of workloads exchange data between threads requiring synchronization and a conduit for transferring data. Synchronization in the form of PTX barriers introduces overhead for NVIDIA architectures by forcing warps to be removed from the scheduling pool until all other warps in a CTA reach the barrier. In fact, some high performance applications including CUDAPP [75] try to reduce these overheads by relying on an implicit barrier between instructions across all threads within a warp – a characteristic of current NVIDIA GPUs, but not of the PTX machine model. This approach breaks compatibility with any architecture whose warp size does not equal that of NVIDIA's (32 threads), but it provides a performance boost on current NVIDIA GPUs.

Providing support for smaller synchronization domains will enable more efficient inter-thread (producer-consumer) data flow - only threads within a domain would be required to wait on a barrier and other threads in a CTA could proceed. Compiler analysis or a local barrier primitive for PTX could allow programmers to express the semantics in a portable way.

3.3.4 Parallelism

Unlike most machine models, PTX programs explicitly declare the number of threads and CTAs in the program, statically exposing parallelism in an application. This parallelism allows programs to be scaled to future GPU architectures simply by adding additional cores and SIMD units. For a given set of applications, it is useful to discover the limits of this scalability in order to determine how many multiprocessors can be used. Towards this end, we define two metrics that capture

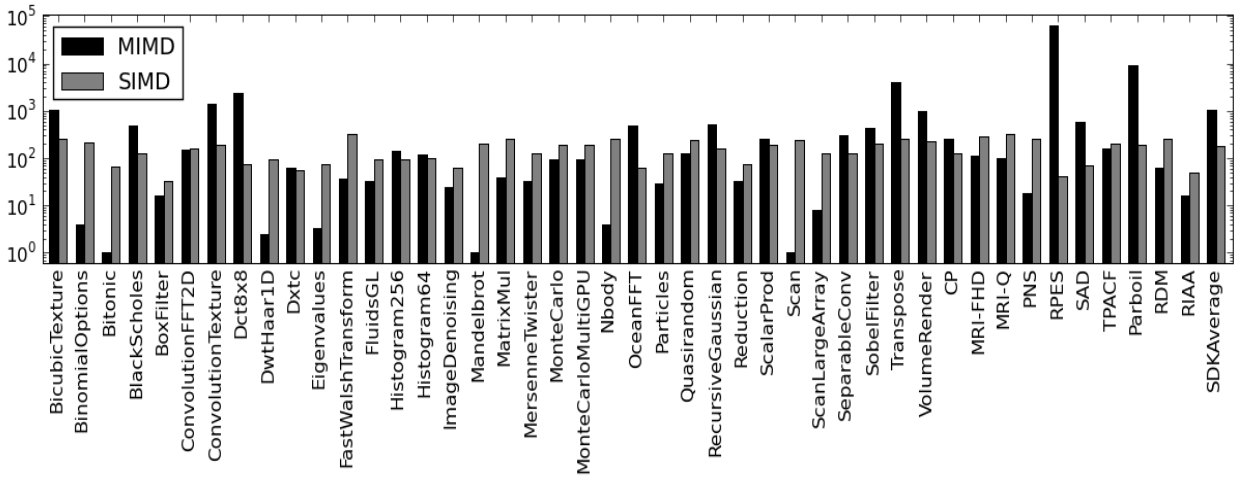


Figure 18: SIMD and MIMD Parallelism

parallelism in an application, *MIMD parallelism* and *SIMD parallelism*.

MIMD parallelism is computed as the speed up of a GPU with an infinite number of multiprocessors over a GPU with a single multiprocessor, ignoring memory bandwidth and latency constraints. It is computed by assuming that each instruction takes a single cycle to complete and dividing the total number of dynamic instructions by the dynamic instruction count (D_i) of the longest running CTA in a kernel as shown in Equation 4. It is averaged over all kernels as in Equation 5. **SIMD parallelism**, on the other hand, is computed as the average activity factor (A_f) of a CTA, multiplied by the number of threads in the CTA, and weighted by the number of dynamic instructions in the CTA as shown in Equation 6. It is averaged over all CTAs in a kernel as in Equation 7.

$$MIMD_{kernel} = \frac{\sum_{i=1}^{ctas} D_i}{\max_{i=1}^{ctas}(D_i)} \tag{4}$$

$$MIMD_{application} = \frac{\sum_{i=1}^{kernels} D_i * MIMD_{kernel\_i}}{\sum_{i=1}^{kernels} D_i} \tag{5}$$

$$SIMD_{kernel} = \frac{\sum_{i=1}^{ctas} A_f * D_i}{\sum_{i=1}^{ctas} D_i} \tag{6}$$

$$SIMD_{application} = \frac{\sum_{i=1}^{kernels} D_i * SIMD_{kernel\_i}}{\sum_{i=1}^{kernels} D_i} \tag{7}$$

Results. The average MIMD parallelism across the SDK is 1076.46, while the average SIMD parallelism is 180.92. This is particularly interesting because it represents a reduced data size for the SDK where many applications are weakly scalable. The Parboil benchmarks, which are significantly larger than the SDK, have a similar SIMD parallelism of 186.96, but an even greater MIMD parallelism of 9427.08. The RDM and RIAA applications have less MIMD parallelism, 62.6 and 15.83 respectively. These applications are also weakly scalable and the lower MIMD parallelism may be an artifact of their higher computational density than many of the SDK examples, several of which simply apply several instruction transformations to each data point. The SIMD parallelism of the SDK, Parboil, and RDM workloads are relatively similar, 180.92, 186.96, and 258.32 respectively, yet strikingly larger than the 49.72 for the RIAA workload. The RIAA workload is limited by a combination of low activity factor and low number of threads per CTA driven by high register usage per thread; the large working set of the application can be seen in the abnormally high memory intensity in Figure 14. Of the 16 CTAs that we did run for this application, the execution time of each CTA was relatively constant resulting in a MIMD parallelism metric of 15.836.

Recommendations. The most important observation that can be drawn from these results is that developers should express PTX programs using as many threads and CTAs as possible to maximize the future scalability of the application. While it is relatively simple for the PTX JIT compiler to serialize CTAs and threads on architectures without many hardware threads [48, 142], it is very difficult to go the other way. A characteristic example can be observed in the Mandelbrot application: in this example, the developer manually compressed the calculation of several pixels

into a single thread. This was most likely done to reduce overheads of starting additional CTAs and using more registers on NVIDIA GPUs, but it ends up limiting the amount of MIMD parallelism in the application by 46.875x.

This practice of artificially reducing the parallelism within an application in order to reduce scheduling and synchronization overheads has been observed in other contexts as well. In a study of Intel's threading building blocks library [36], it was observed that adding too many parallel tasks in a Black Scholes application caused synchronization costs to overwhelm any performance gained from parallelism. The fact that CTAs in PTX can be serialized by the compiler alleviates this problem to some extent, but this requires intelligent compiler heuristics that balance between parallelism and overheads. In general there is unresolved contention between optimizations that reduce overheads and optimizations that improve parallelism. This is the subject of execution model translation described in Chapter 5.

3.4 Concluding Remarks

The result of this study answers some fundamental questions regarding this type of execution model and the nature of applications targeting it. Many applications exhibit some form of thread divergence that must be accommodated. Dynamic instruction counts are highly sensitive to warp formation and thread scheduling which motivates research in developing efficient thread reconvergence policies. Application performance is greatly impacted by memory efficiency, and many applications do not achieve peak effieciency implying an avenue of possible optimizations. The amount of parallelism within these applications is considerable, on average scaling to over 1000 cores, each executing tens or hundreds of threads with fine-grain synchronization. This degree of data-parallelism overcomes the fundamental limiting factor described in [54]. Finally, many applications exhibit some degree of synchronization among threads implying there must be efficient resources capable of retaining live state.

The metrics defined in Section 4.3.1 capture properties about applications that will be later applied in subsequent chapters of this thesis. Chapter 4 describes a statistical performance modeling technique that utilizes the results of these metrics and illustrates their usefulness in predicting application performance. Chapter 5 applies insights and observations about activity factor, memory efficiency, and SIMD scalability to designing and implementing a dynamic compilation technique for translating the PTX execution model for heterogeneous CPU architectures. Chapter 6 explores region-based compilation and scheduling for improving control-flow uniformity and considers the results of these metrics as well as program structure as inputs to partitioning heuristics. Finally, Chapter 7 evaluates the performance of GPU Ocelot's various device backends as well as novel thread reconvergence techniques in terms of these metrics.

CHAPTER IV

PERFORMANCE MODELING

This chapter describes a statistical performance modeling approach and an automated infrastructure implementing the same to construct performance models based on *a priori* profiling results, machine and application parameterization, and model selection. Evaluating this technique reveals several non-intuitive relationships between program characteristics and performance. This study also demonstrates *a priori* and initialization-time metrics may be sufficient to predict performance trends before the kernel is executed.

4.1 Introduction

Detailed architectural simulations of compute nodes within a distributed or cluster environment experience an exponentially increasing growth of simulation time as Moore's Law scales the complexity of microprocessors. Limitations in current techniques for parallel simulation have yielded poor performance scaling. Thus, the capacity to perform detailed timing simulation for design space exploration of processors intended for large scale applications is limited. Design space explorations on real-world benchmarks may be performed by simulating a single node while the interface between that node and its cluster environment is simualted at a coarse-grain level. Simulating an approximation of this environment is not, however, trivial. Analytical models must be specialized for individual applications and datasets and may not adapt to changed parameters in the simulated platform or network.

Alternatively, a profile-driven statistical approach to performance modeling enables the rapid creation and deployment of performance models without the need for detailed analysis of individual workloads. This chapter presents a performance modeling approach to semi-automatically

construct a statistical performance model of heterogeneous workloads. This model is trained on profiling results acquired through instrumenting the CUDA Runtime, executing applications on a detailed functional simulator, and acquiring real-world performance results on actual hardware. The results of concurrent profiling runs are composited in a database. Our results show improved accuracy with clustering analysis. Finally, a model selection phase determines the statistical function of the profiling data, projected onto principal component space, that minimized total squared error. The resulting performance model may be used to estimate runtime or other desired performance metrics as execution parameters are varied.

As more applications are written from the ground up to perform efficiently on systems with a diverse set of available processors, choosing the architecture capable of executing each kernel of the application most efficiently becomes more important in maximizing overall throughput and power efficiency. At the time of this writing, we are not aware of any study that correlates detailed application characteristics with actual performance measured on real-world architectures.

Consequently, this chapter describes the following contributions:

- We identify several PTX application characteristics that indicate relative performance on GPUs and CPUs.
- We use a statistical data analysis methodology based on principal component analysis to identify critical program characteristics.
- We introduce a model for predicting relative performance when the application is run on a CPU or a GPU.

The analysis and models presented in this chapter leverage the GPU Ocelot framework for instrumenting data-parallel applications and executing them on heterogeneous platforms. Specifically, this evaluation leverages GPU Ocelot's emulation backend for detailed application profiling as well as the multicore CPU and the NVIDIA GPU backends for executing applications at full

speed on target processors. Ocelot is uniquely leveraged to gather instruction and memory traces from emulated kernels in unmodified CUDA applications, analyze control and data dependencies, and execute the kernel efficiently on both CUDA-capable GPUs and multicore CPUs. Consequently, in addition to enabling detailed and comparative workload characterizations, the infrastructures enables transparent portability of PTX kernels across CPUs and NVIDIA GPUs. The techniques described in this chapter were first published in [96] and extended in [**?**].

As programming models such as NVIDIA's CUDA [124] and the industry-wide standard OpenCL [66] gain wider acceptance, efficiently executing the expression of a data-parallel application on parallel architectures with dissimilar performance characteristics such as multi-core superscalar processors or massively parallel graphics processing units (GPUs) becomes increasingly critical. While considerable efforts have been spent optimizing and benchmarking applications intended for processors with several cores, comparatively less effort has been spent evaluating the characteristics and performance of applications capable of executing efficiently on both GPUs and CPUs.

High-performance computing presents system-level and microarchitecture-level design problems with numerous trade-offs to be made with significant and complex interactions on the resulting performance and efficiency of software. Simulation presents a solution approach, enabling the evaluation of massively parallel systems and justification for design choices. However, simulation is typically much slower than the systems being simulated, and does not exhibit the same performance scaling with concurrency. Coarse-grained simulation enables selectively performing detailed simulation of an application executing on a single node while skeletons and performance models simulate its interactions with neighboring nodes. SST/Macro [86] provides an interface for defining application skeletons which still require correct and faithful modeling of message rates, communications latency, runtimes, and energy. By reducing the complexity of simulating these coarse-grained processes through modeling, the overall simulation workload is significantly reduced without substantially impacting accuracy. This is particularly apparent during design space exploration in which application skeletons driven by performance models enable fast simulation.

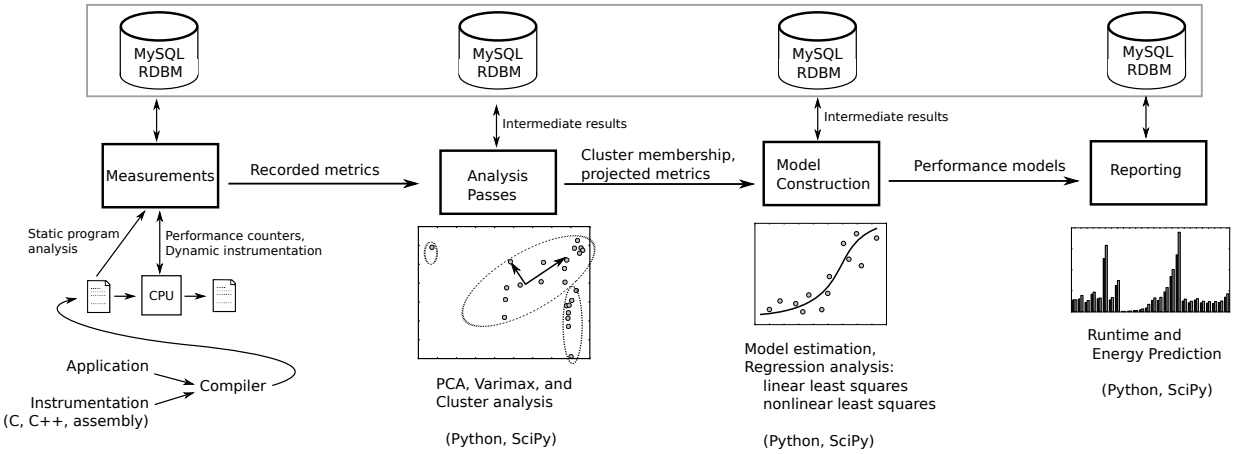


Figure 19: Implementation details of the Eiger Statistical Model Creation framework.

4.2 Modeling Technique

This section describes elements of the proposed performance modeling infrastructure, collectively referred to as "Eiger." A detailed illustration is provided in Figure 19. The following components constitute an automated process in which application profiling data is collected and ultimately used to construct a model of runtimes, energy, or any other dependent result metric. The resulting statistical model may be then composed with other tools and applications such as simulation environments, heterogeneity-aware process schedulers, and reporting tools. This chapter describes the construction of models of execution time. An application is executed on the target processor, multiple parameters recorded, and execution time measured. This is defined as one trial. Multiple trials for an application cover different parameter values.

Distributed Profiling Acquisition Due to the mutually independent nature of each trial, the individual runs required to accumulate this information can intrinsically be run independently. We use a relational database to manage the storage of all data accumulated during profiling runs as it allows asynchronous insertions while allowing for rigid relational specifications. Additionally, we provide for the scenario where runs of multiple tools are required to construct a single data point (trial). This is the case, for example, when the instrumentation to collect application metrics

interfere with the performance of the application, in which case the application would have to be run a second time to collect the performance results.

Dimensionality Reduction. The design supports the addition of analysis passes. We will start with Principal Component Analysis (PCA) which is a well-known dimensionality reduction technique. The major computations are construction of a correlation matrix and computing the eigenvectors of this correlation matrix via Singular Value Decomposition (SVD) [64]. Implementations of SVD are available in LAPACK implementations, notably SciPy [3]. Benefits of reducing the dimensionality of the input dataset are manifold; it speeds up the model generation process, improves the clarity of the resulting model, and allows for intuition into the correlations between input metrics. It is important to note that PCA is an *unsupervised* learning technique in that it does *not* take the performance metric into account when choosing dimensions to eliminate. It is entirely possible that a dimension with low variance may have a larger affect on application performance than one with high variance. For example, number of cores may not have as large a variance as memory bandwidth for a range of machines but a greater impact on runtime for compute-bounded applications. PCA does not explicitly specify how many dimensions to retain; rather it relies upon the user to make the final decision. One popular technique is to use a *scree graph*, which charts the amount of variance captured by each dimension. The user is then able to choose a suitable number of components after which the captured variance drops off significantly. The scree graph for this work appears in Figure 20.

Cluster Analysis. Designing processors to accelerate general sets of workloads would be significantly simpler if all workloads exhibited similar performance characteristics. In contrast, real applications demonstrate varied performance behavior. Dense linear algebra workloads with regular control properties and compute-intensive inner loops differ significantly from irregular workloads with data-dependent branch behavior and load imbalance across threads. Goswami et al. [65] analyze the diversity of CUDA workloads and present a prioritized tree of benchmark applications

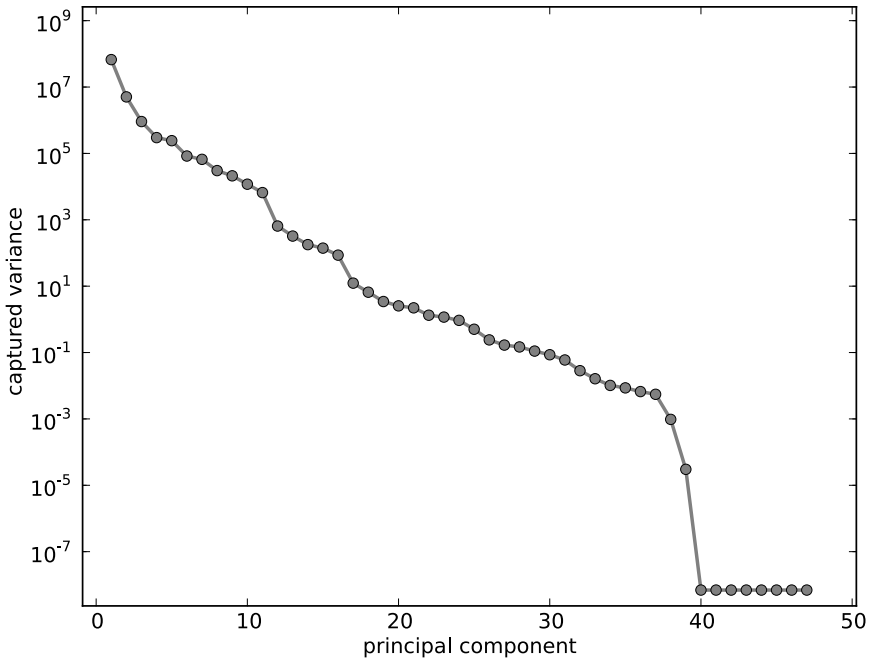


Figure 20: Example scree graph for selecting dimensions to eliminate.

sorted by how much each increases total variance of a given set of metrics. Applications exhibiting high correlation among principal components are clustered together. A single model trained from profiling data from all applications in a comprehensive benchmark is unlikely to yield high accuracy. Rather, the best model is likely to be obtained from training data gathered by a similar set of applications to the experimental application. Kerr et al. [96] provide empirical evidence that clustering and partitioning improves model accuracy.

Model Selection and Training. Model construction is a pass over the data to produce a model. Our first model construction pass will be automated regression analysis [133] which constructs an analytic model determined from a set of training samples. In this case, the samples are taken from data projected onto the principle component basis vectors. Regression modeling yields an analytic formula for computing runtime and/or energy from additional signals. PCA and varimax satisfy several assumptions enabling classical regression analysis techniques. Principal Component

Analysis yields orthogonal and uncorrelated principal components. The inputs are determinstic and considered free from error. We propose parametric regression models.

Each step in the stepwise procedure, shown in Algorithm 4, considers the function from the model pool that, if added to the current model, would increase the fit the most. We define the coefficient of determination, $\bar{R}^2$, as follows:

$$R^2 = 1 - \frac{\sum_i (y_i - f_i)^2}{\sum_i (y_i - \bar{y})^2}$$

If the coefficient of determination, $\bar{R}^2$, of the new model, including the candidate function, surpasses the $\bar{R}^2$ of the model without the new function by more than the provided *threshold*, the function is added to the model, removed from the model pool, and the algorithm starts again. When there are no more functions remaining in the model pool that would pass this threshold, the algorithm completes and returns the final model. This final model may not have maximally reduced squared error for the training data, but the formulation of $\bar{R}^2$ provides for a model that is more likely to predict values not present in the training set.

Model Pool The model pool defines a set of possible functions which may be mapped to principal components and whose linear combination yields the resulting performance model. The model pool must be selected by the experimenter and should offer sufficient variety for maximizing goodness of fit of the resulting model. The model pool should include functions that closely model the space and time complexity of dominant algorithms within the applications of interest as well as non-linear combinations of several metrics. For example, compute-bound applications may demonstrate a very strong correlation between the product of clock frequency and dynamic instruction counts. Table 5 describes the selection of functions used in the model pool for this work.

Reporting Completed models consist of a set of transformation matrices from dimensionality reduction and cluster analysis as well as a vector of functions and their associated weights. Reporting passes over this data format the information in a method easily consumed by the user, including

Table 5: Model pool.

Functions							
x_i^{-2}	x_i^{-1}	$x_i^{-1/2}$	$log_2(x_i)$	$x_i^{1/2}$	x_i^1	x_i^2	$x_i * x_j$

plotting and statistical results. This phase also allows for the serialization and memoization of the finished models for later consumption.

4.2.1 Formal Specification

Let $M \in Z$ refer to the number of *trials* executed for a multiplicity of applications, datasets, and machine configurations. Let $N \in Z$ refer to the total number of metrics per trial acquired. These may include static application metrics, dynamic metrics acquired during the execution of the trial, and machine configuration parameters.

We define $X \in R^{m \times n}$ as an input dataset and $R \in R^{m \times 1}$ as a result set. Each row in X corresponds to a trial instance with result in the corresponding row of the column vector R. Together, (X, R) captures sufficient data describing the application, machine, and performance characteristics to construct a model for R. The tuple (X, R) captures the complete profile of applications and their results. Principle component analysis (PCA) yields a projection P from X onto $U' \in R^{m \times p}$, where p is the number of principle components.

$$X = \begin{bmatrix} \vdots & & \\ M \text{ trials} & & \\ \vdots & & \\ \dots & N \text{ metrics} & \dots \end{bmatrix} \qquad R = \begin{bmatrix} \vdots \\ M \text{ results} \\ \vdots \end{bmatrix}$$

Clustering analysis enables a down-selection of trials for computing particular models for a set of applications. This analysis yields a subset of rows such that $U = SU'$ where $U \in R^{m' \times p}$ and S

is a selection matrix. This work applies k-means clustering which partitions a set of points into k clusters such that each point in a cluster k_i is closest to the mean of k_i than any other mean. The distance metric used in this work is *squared Euclidean distance*.

Model selection yields a function $f : R^{1\times p} \rightarrow R$ that maps individual trials onto a predicted result value. f is the performance model, and this work yields one model per cluster. Model selection leverages linear regression, a commonly-used and well-behaved form of regression that evaluates the dependent variable y as the weighted linear combination of independent variables x plus an error term ϵ, representing any deviation of the expected value of the model from the real value.

$$y = \beta_0 + \sum_{i=1}^{n} \beta_i x_i + \epsilon \tag{8}$$

The method of model estimation is least squares, in which the set of coefficients β is chosen to minimize the residual sum of squares

$$RSS(\beta) = \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij} B_j)^2 \tag{9}$$

Model selection itself composes a performance model as the linear combination of a set of non-linear functions on the projected profiling data. The set of possible functions is known as a *model pool*, which can include basis expansions, mathematical transformations, and variable interactions, among others. To allow for greater generalizability, this work can iterate over a set of model pools and select the one that minimizes squared error.

In order to manage model complexity and optimize training error rate, a forward-stepwise procedure is used to aggregate basis functions based upon adjusted coefficient of determination, a modification of the coefficient of determination that adjusts for the number of terms in the model.

$$Adjusted R^2 = \bar{R}^2 = 1 - (1 - R^2)\frac{n-1}{n-p-1} \tag{10}$$

It is important to note that $\bar{R}^2$ does not have the same interpretation as R^2; while R^2 is in the range $0 \rightarrow 1$, $\bar{R}^2$ is in the range $-\infty \rightarrow 1$. The intuition is that any value of $\bar{R}^2$ less than zero implies a fit worse than could be expected by chance.

```
profile = ... // initialize profile matrix
performance = ... // initialize performance vector
modelPool = ... // initialize model pool
finalModel = []
currentRsquared = −∞
begin
    while not done do
        foreach each function left in modelPool do
            append function to finalModel
            U = apply finalModel to profile
            beta = leastSquares(U, performance)
            Rsquared = ... // calculate adjusted rsquared
            if Rsquared ¿ maximum then
                currentMax = Rsquared
                newFunction = function
                newBeta = beta
            end
            remove current function from finalModel
        end
        if currentMax - currentRsquared ¿ threshold then
            append newFunction to finalModel
            remove newFunction from modelPool
            currentRsquared = currentMax
        end
        else
            done = True // there are no more useful functions
        end
    end
end
return finalModel, beta
```

Algorithm 4: Selects a model function that minimizes error over a cluster

4.3 Characterization Methodology

4.3.1 Metrics and Statistics

Ocelot's PTX emulator may be instrumented with a set of user-supplied event handlers to generate detailed traces of instructions and memory references. After each dynamic PTX instruction is completed for a given program counter and set of active threads, an event object containing program counter, PTX instruction, activity mask, and referenced memory addresses is dispatched to each registered trace generator which handles the event according to the performance metric it implements. We present the following application metrics gathered in this manner building on the set of metrics defined in our previous work [95]:

Activity Factor. Any given instruction is executed by all threads in a warp. However, individual threads can be predicated off via explicit predicate registers or as a result of branch divergence. Activity factor is the fraction of threads active averaged over all dynamic instructions.

Branch Divergence. When a warp reaches a branch instruction, all threads may branch or fall through, or the warp may diverge in which the warp is split with some threads falling through and other threads branching. Branch Divergence is the fraction of branches that result in divergence averaged over all dynamic branch instructions.

Instruction Counts. This metric counts the number of dynamic instructions binned according to the functional unit that would execute them on a hypothetical GPU. The functional units considered here include integer arithmetic, floating-point arithmetic, logical operations, control-flow, off-chip loads and stores, parallelism and synchronizations, special and transcendental, and data type conversions.

Inter-thread Data Flow. The PTX execution model includes synchronization instructions and shared data storage accessible by threads of the same CTA. Interthread data flow measures the fraction of loads from shared memory such that the data loaded was computed by another thread within the CTA. This is a measure of producer-consumer relationships among threads.

Memory Intensity. Memory intensity computes the fraction of instructions resulting in communication to off-chip memory. These may be explicit loads or stores to global or local memory, or they may be texture sampling instructions. This metric does not model the texture caches which are present in most GPUs and counts texture samples as loads to global memory.

Memory Efficiency. Loads and stores to global memory may reference arbitrary locations. However, if threads of the same warp access locations in the same block of memory, the operation may be completed in a single memory transaction; otherwise, transactions are serialized. This metric expresses the minimum number of transactions needed to satisfy every dynamic load or store divided by the actual number of transactions, computed according to the memory coalescing protocol defined in [124] §5.1.2.1. This is a measure of spatial locality.

Memory Extent. This metric uses pointer analysis to compute the working set of kernels as the number and layout of all reachable pages in all memory spaces. It represents the total amount of memory that is accessible to a kernel immediately before it is executed.

Context Switch Points. CTAs may synchronize threads at the start and end of kernels as well as within sections of code with uniform control flow, typically to ensure shared memory is consistent when sharing data. Each synchronization requires a context switch point inserted by Ocelot during translation for execution on multicore as described in [49].

Live Registers. Unlike CPUs, GPUs are equipped with large register files that may store tens of live values per thread. Consequently, executing CTAs on a multicore x86 CPU requires spilling values at context switches. This metric expresses the average number of spilled values.

Machine Parameters. GPUs and CPUs considered here are characterized by clock rate, number of concurrent threads, number of cores, off-chip bandwidth, number of memory controllers, instruction issue width, L2 cache capacity, whether they are capable of executing out-of-order, and the maximum number of threads within a warp.

Registers per Thread. The large register files of GPUs may be partitioned into threads at runtime according to the number of threads per CTA. Larger numbers of threads increases the

Table 6: Benchmark Applications.

Application	Full Name	Source
MRI-Q	Magnetic Resonance Imaging	Parboil
MRI-FHD	Magnetic Resonance Imaging	
CP	Coulombic Potential	
SAD	Sum of Absolute Differences	
TPACF	Two-Point Angular Correction	
PNS	Petri Net Simulation	
RPES	Rys Polynomial Equation Solver	
hotspot	Thermal simulation	Rodinia
lu	Dense LU Decomposition	
nbody	Particle simulation	CUDA SDK

ability to hide latencies but reduces the number of registers available per thread. On CPUs, these may be spilled to local memory. This metric expresses the average number of registers allocated per thread.

Kernel Count. The number of times an application launches a kernel indicates the number of global barriers across all CTAs required.

Parallelism Scalability. This metric determines the maximum amount of SIMD and MIMD parallelism [95] available in a particular application averaged across all kernels.

DMA Transfer Size. CUDA applications explicitly copy buffers of data to and from GPU memory before kernels may be called incurring a latency and bandwidth constrained transfer via the PCI Express bus of the given platform. We measure both the number of DMAs and the total amount of data transferred.

4.3.2 Benchmarks

For this study, we selected applications from existing benchmark suites. PARBOIL [81] consists of seven application-level benchmarks written in CUDA that perform a variety of computations including ray tracing, finite-difference time-domain simulation, sorting. Rodinia [26] is a separate collection of applications for benchmarking GPU systems. Finally, the CUDA SDK is distributed

with over fifty applications showcasing CUDA features. A list of the applications we selected appears in Table 6.

Kernels from these applications were executed on processors whose parameters are summarized in Table 7. This selection consists of both CPUs and GPUs that together offer a wide range for each of the listed parameters. We expect these parameters that capture clock frequency, issue width, concurrency, memory bandwidth, and cache structure to sufficiently model the performance of kernels from the benchmark applications.

We chose to characterize PTX applications by the collection of statistics listed in Table 8. These may be classified according to the way they are gathered. Some quantities may be determined via *static analysis* before a kernel is executed such as static instruction counts of each kernel, the number/size of DMA operations initiated before a kernel launch, as well as upper bounds on working set size determined by conservative pointer analysis. Others may be determined at runtime by inserting *instrumentation* into kernels and recording averages as they execute; these include SIMD and MIMD parallelism metrics. Finally, some metrics – typically dynamic instruction counts – may only be determined by executing the kernel to completion via PTX *emulation* and analyzing the resulting instruction traces. Note that all of these metrics were collected via execution on the Ocelot PTX Emulator, therefore, they are independent of the micro-architecture of a particular CPU or GPU. Table 8 lists the quantitative results from each Parboil application.

4.4 Results

Our methodology for modeling the interaction between machine and program characteristics uses principal component analysis to identify independent parameters, similar to [53], cluster analysis to discovers sets of related applications, and multivariate regression combined with projections onto convex sets [17] to build predictive models from principal components.

Table 7: Machine parameters.

	Nehalem	Atom	Phenom	8600 GS	8800 GTX	GTX280	C1060
Type	Out-of-order CPU	In-order CPU	Out-of-order CPU	In-order GPU	In-order GPU	In-order GPU	In-order GPU
Issue Width	4	2	3	1	1	1	1
Clock Frequency (GHz)	2.6	1.6	2.2	1.2	1.5	1.3	1.3
Hardware Threads per Core	2	2	1	24	24	24	24
Cores	4	1	4	2	16	30	30
Warp Size	1	1	1	32	32	32	32
Memory Controllers	3	1	2	2	6	8	8
Bandwidth per Controller (GB/s)	8.53	3.54	8.53	5.6	14.3	17.62	12.75
L2 Cache (kB)	512	512	512	0	0	0	0

4.4.1 Principal Component Analysis

Principal Component Analysis (PCA) is predicated on the assumption that several variables used in an analysis are correlated, and therefore measure the same property of an application or processor. PCA derives a set of new variables, called principal components, from linear combinations of the original variables such that there is no correlation between any of the new variables. PCA identifies the new variables with the most information about the original data thereby reducing the total number of variables needed to represent a data set. In our analysis, we use a normalized PCA (zero mean, unit variance) because each of our original metrics are expressed using different units. We choose enough principal components to account for at least 85% of the variance in the original data.

Once PCA has identified a set of principal components, we apply a varimax rotation [114] to the principal components. This distributes the contribution of each original variable to each principal component, such that each original variable either strongly impacts a principal component or it very weakly impacts it. In other words, it causes each original metric to influence a single principal component, easing analysis of the data.

For the statistics gathered in the previous section, we perform two separate PCAs: one which

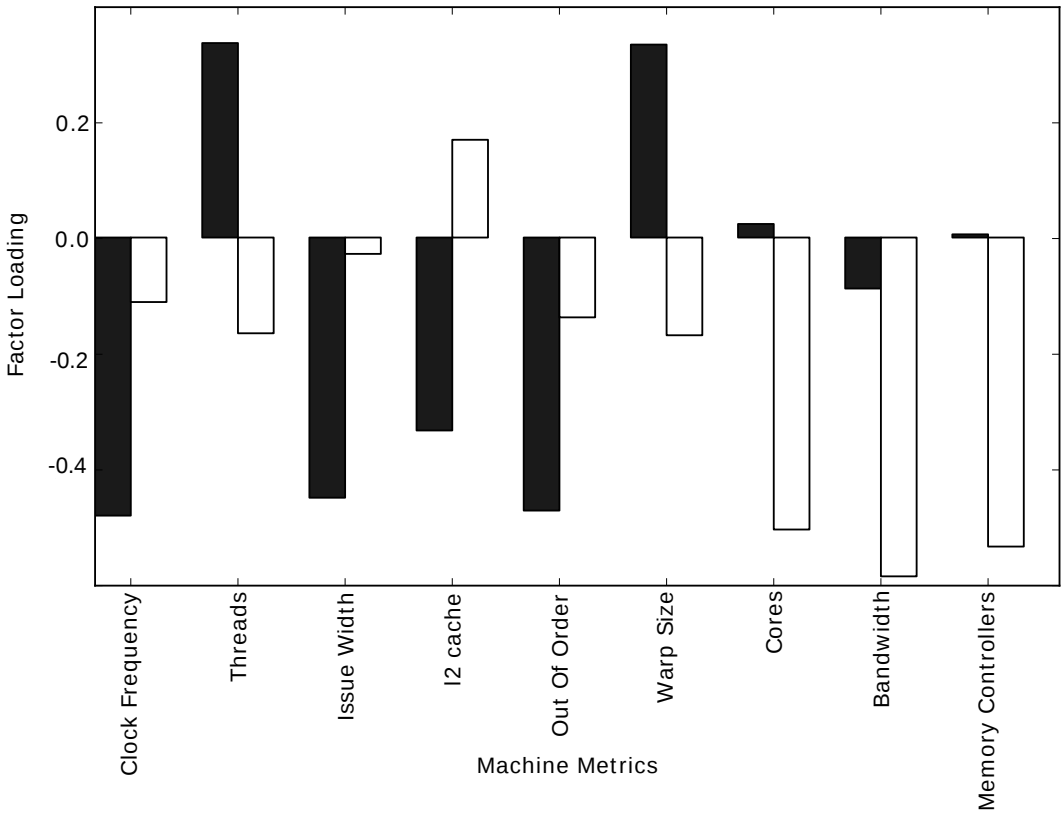


Figure 21: Factor loadings for two machine principal components. PC0 (black) corresponds to single core performance, while PC1 (white) corresponds to multi-core throughput.

only includes application statistics and another which only includes machine statistics. This is valid because the metrics were collected via the Ocelot PTX emulator, which is architecture agnostic.

4.4.2 Machine Principal Components

From the set of machine statistics, PCA yielded two principal components that are shown in terms of factor loadings in Figure 21 and plotted in Figure 22. Clusters reiterate the few number of high-speed cores among the CPUs and a much larger number of lower-speed cores among the GPUs.

PC0: Single Core Performance. The variables that contribute strongly to the first principal component are shown in the left of Figure 21. Note that all of these metrics, clock frequency, issue

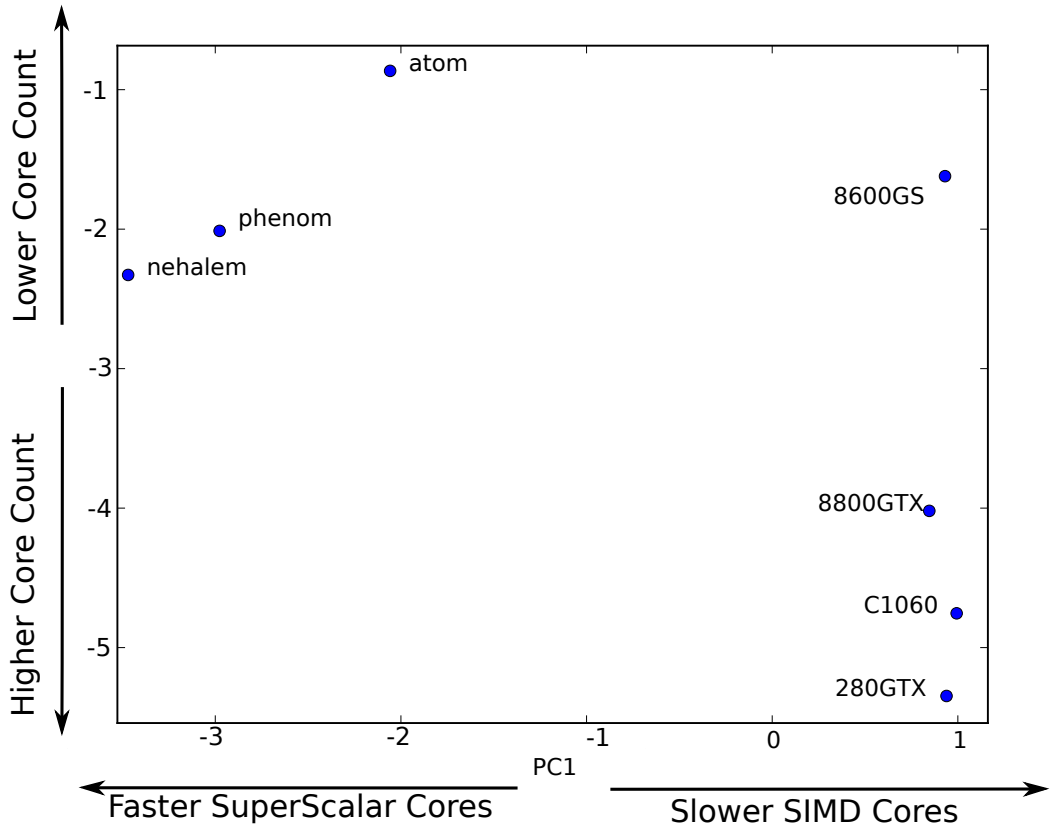


Figure 22: The machine principal components. GPUs have high core counts and slow SIMD cores while CPUs have fewer, but faster, cores.

width, cache size, etc correspond to the performance of a single processor core. Additionally, note that threads-per-core and warp size are negatively correlated with clock frequency, issue width, and out of order, highlighting the differences between GPU and CPU design philosophies.

PC1: Core and Memory Controller Count. The second PC illustrates that the core count is correlated with the memory controller count and memory bandwidth per channel, indicating that multi-core CPUs and GPUs are designed such that the off-chip bandwidth scales with the number of cores.

Discussion. Though the intent of this chapter is to derive relationships between these metrics and the performance of machine-application combinations, this analysis also exposes trends in the way that CPUs and GPUs are designed. The division of the machine metrics into these two

principal components can be explained as follows: the design of a single core typically does not influence the synthesis of many single cores and memory controllers into a multi-core processor. The performance of a single core in a processor is either characterized by clock frequency, cache size, superscalar width, etc or a high degree of hardware multithreading and large SIMD units. Figure 22 shows a clear distinction between CPU and GPU architectures.

This classification holds even for processors not included in this study. For example, recently released GPUs by Intel [82] and AMD can both be characterized by a large number of threads per core and wide SIMD units; even embedded CPUs such as ARM Cortex A9 [39] have begun to move towards out of order execution.

4.4.3 Application Components

The PCA of the application statistics yielded five principal components, the factor loadings of which are shown in Figure 23. We would like to note that PCA reveals relationships that hold only for a given set of data, in this case, the applications that were chosen. Given a different set of applications, PCA may reveal a different set of relationships. However, the fact that these trends are valid across applications from both Parboil and Rodinia, which are designed to be representative of CUDA applications, indicates that they may represent fundamental similarities in the way that developers write CUDA programs.

PC0: MIMD Parallelism. The first principal component is composed of metrics that are related to the MIMD parallelism of a program. Recall that MIMD parallelism measures the speedup of a kernel on an idealized GPU with an infinite number of cores and zero memory latency. It is bound by the number of CTAs in each kernel. The correlation between MIMD parallelism and DMA Size indicates that applications that copy a larger amount of memory to the GPU will also launch a large number of CTAs. It is interesting to note that during our preliminary evaluation which only included the Parboil benchmarks shown in Table 8, this component also included the majority of dynamic instruction counts. Adding the Rodinia *hotspot* application and the SDK

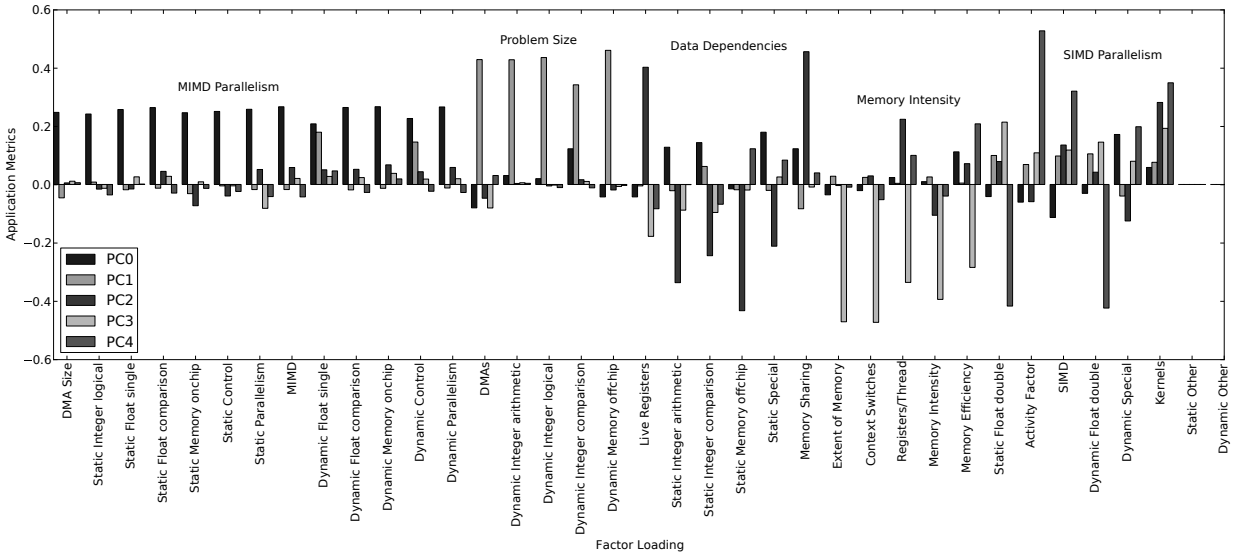


Figure 23: Factor loadings for the five application principal components. A factor loading closer to +-1 indicates a higher influence on the principal component.

nbody application broke the relationship between MIMD parallelism and problem size, indicating that not all CUDA applications are weakly scalable. This distinction motivates the need for the cluster analysis in the next section, where applications with similar characteristics can be identified and modeled separately. As a final point, notice that several static instruction counts are highly correlated with the problem size. This relationship is difficult to explain intuitively, and it would be relatively simple to craft a synthetic application that breaks this relationship. However, it is significant that none of these applications do. Results like this motivate the use of a technique like PCA, which is able to discover relationships that defy intuition.

PC1: Problem Size. The second component is composed most significantly of average dynamic integer, floating point, and memory instruction counts which collectively describe the number of instructions executed in each kernel. As described in the analytical model developed by Hong et. al. [79], these dynamic instruction counts are strong determinants of the total execution time of a program, and therefore the high degree of correlation is expected. What is not obvious is the relationship between the number of DMA calls executed before a kernel is launched and these

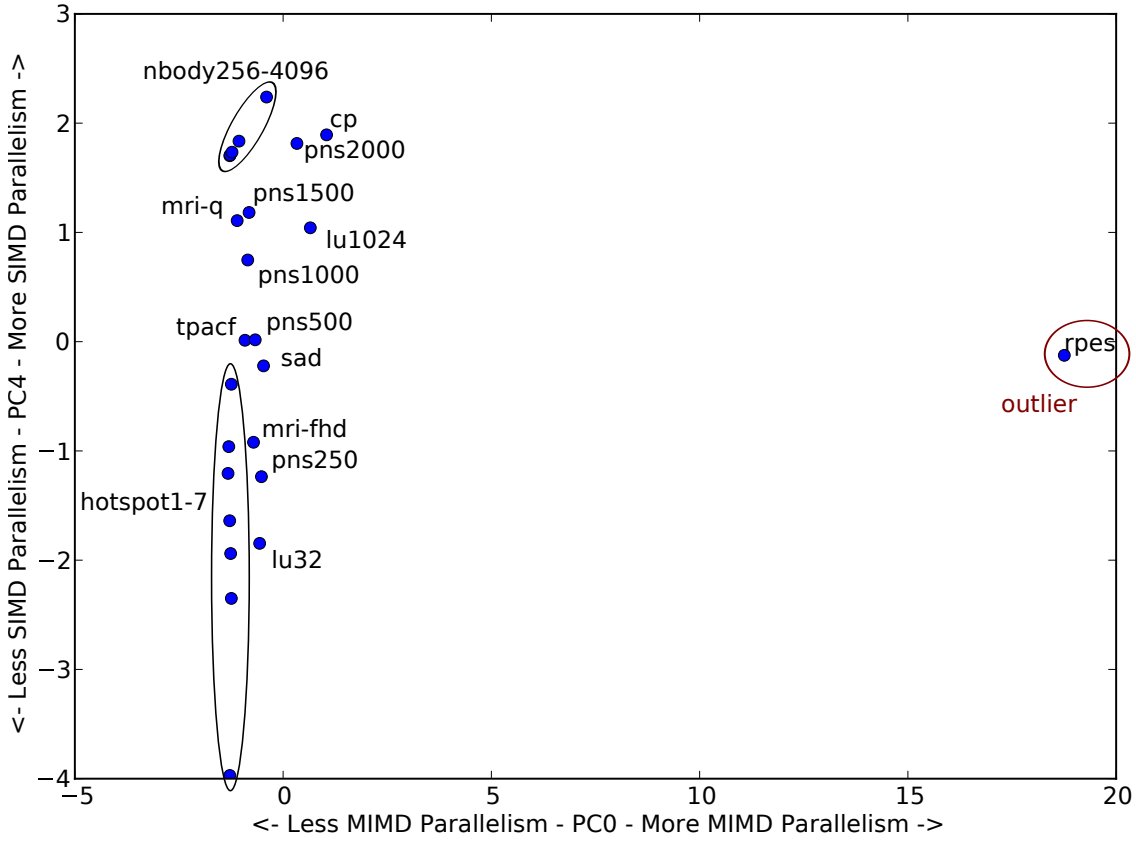


Figure 24: This plot compares MIMD to SIMD parallelism. It should be clear that these metrics are completely independent for this set of applications; the fact that an application can be easily mapped onto a SIMD processor says nothing about its suitability for a multi-core system. A complementary strategy may be necessary that considers both styles of parallelism when designing new applications.

instruction counts. We find that across all principal components, there is at least one metric that is available before launching a kernel that is highly correlated with the dynamic metrics in that component. We exploit this property in Section 4.4.5 to build a predictive model for application execution time using only static metrics.

PC2: Data Dependencies. We believe that the second principal component exposed the most significant and non-obvious relationship in this study. It indicates that data dependencies are likely to be propagated throughout all levels of the programming model; if there is a large degree of data sharing between instructions, then there is likely to be a large degree of data sharing among

threads in each CTA and among all CTAs in a program. Notice that this component shows that registers that are alive at context switch points, memory sharing, and total kernel count are highly correlated. Data is typically passed from one thread to another at context switch points. More context switch points imply more opportunities for sharing data between threads and more registers alive at these context switch points imply more data that can be transferred to another thread. Memory sharing measures exactly the amount of memory that is passed from one thread to another through shared memory. Finally, CTAs cannot reliably exchange data within the same kernel, but kernels have implicit barriers between launches that allow data to be exchanged between CTAs in different kernels. The correlation between memory sharing and kernel count seems to indicate that programmers will break computations that are required to share data among CTAs into multiple kernels. Furthermore, it seems to indicate that programs are either embarrassingly parallel at all levels, from the instruction level up to the task level, or have dependencies at all levels.

PC3: Memory Intensity. The next principal component is composed almost entirely of metrics that are associated with the memory behavior of a program. It should be clear that kernels that are given access to a large pool of memory via pointers are likely to access a significant amount of it. It is also interesting that applications that access a large amount of memory are likely to access it relatively efficiently, possibly because it can be accessed in a streaming rather than a random pattern. This component reveals that the memory intensive nature of applications is reflected in all levels of the memory hierarchy, from the register pressure to the ratio of memory to compute instructions to the amount of memory accessible by a kernel; for this analysis, a program with high register pressure can be predicted to be very memory intensive. Finally, this component is negatively correlated with dynamic floating point instruction count, indicating that applications either stress the memory hierarchy or the floating point units in a given processor, but not both. This information could be used in the design of highly heterogeneous architectures where some processors are given low latency and high bandwidth memory links, others are given extra floating point units, and workloads are characterized and directed to one or the other accordingly.

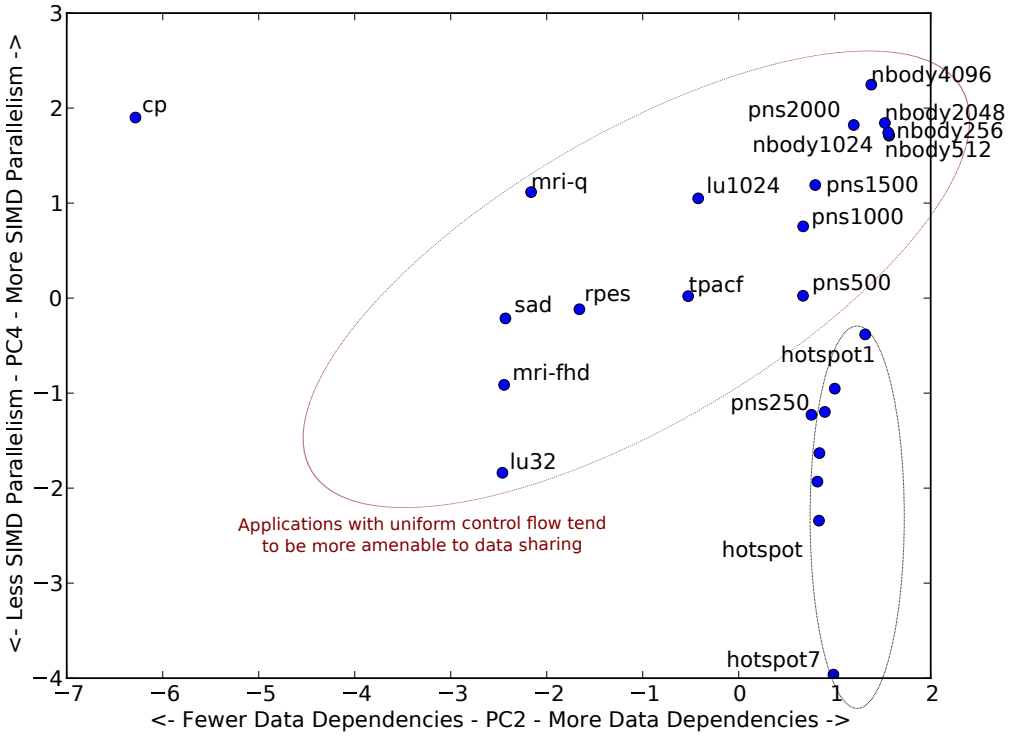


Figure 25: A comparison between Control Flow Divergence and Data Dependencies/Sharing. Excluding the hotspot applications, applications with more uniform control flow exhibit a greater degree of data sharing among threads. Well structured algorithms may be more scalable on GPU-like architectures that benefit from low control flow divergence and include mechanisms for fine grained inter-thread communication.

PC4: Control Flow Uniformity/SIMD Parallelism. The final component exposes several very interesting relationships involving the Activity Factor of an application. Recall that Activity Factor refers to the average ratio of threads that are active during the execution of a given dynamic instruction. First, Activity Factor is directly correlated with special instructions, indicating that it is unlikely that texture or transcendental operations will be placed immediately after divergent branches; if a special instruction is executed by one thread, it is likely to be executed by all other threads. This relationship is reversed for double precision floating point instructions; programs that execute a significant number of double precision instructions are likely to be highly divergent.

Discussion. Though the intent of this analysis was to identify uncorrelated metrics that could be used as inputs to the regression model and cluster analysis in the following sections, PCA also

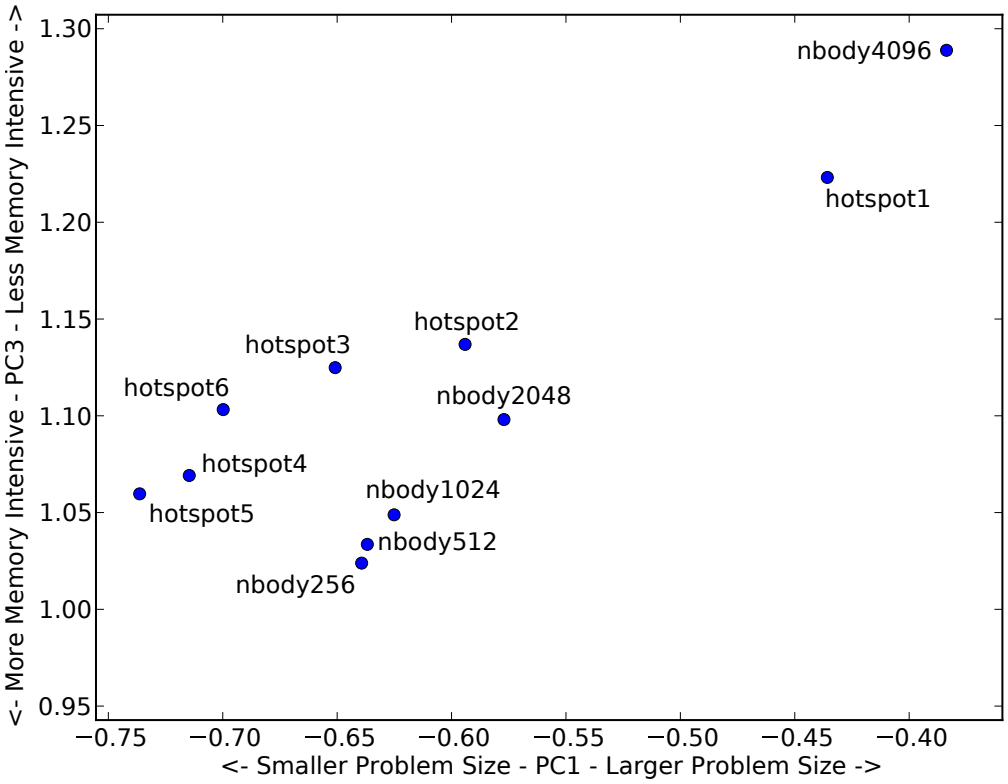


Figure 26: This figure shows the effect of increased problem size on the Memory Intensity of the Nbody and Hotspot applications. While this relationship probably will not hold in general, it demonstrates the usefulness of our methodology for characterizing the behavior of individual applications. We had originally expected these applications to become more memory intensive with an increased problem size; they actually become more compute intensive. This figure is also useful as a sanity check for our analysis, it correctly identifies the Nbody examples with higher body counts as having a larger problem size.

exposed several key relationships between program characteristics that may inform the design of CUDA applications, data parallel compilers, or even new processors optimized for different classes of applications. For example, we expected the dynamic single precision floating point instruction count to be negatively correlated with the dynamic double precision count. However, as can be seen in Figure 23, they are not correlated at all, indicating that many applications perform mixed precision computation. After examining several applications, we realized that some used floating point constants in expressions involving single precision numbers. The compiler interprets all

floating point constants as double precision unless they are explicitly specified to be single precision, and any operations involving these constants would be cast up to double precision, performed at full precision, and then truncated and stored in single precision variables. This is probably not the intention of the developer, and in processors where there are limited double precision floating point units, such as the C1060, this may incur a significant performance overhead.

4.4.4 Regression Modeling

The goal of this study is to derive accurate models for predicting the execution time of CUDA applications on heterogeneous processors. If possible we would also like to be able to make these predictions using only metrics that are available before a kernel is executed. As shown in Section 4.4.3, 85% of the variance across the set of metrics can be explained by five principal components, each of which is composed of at least one static metric that is available before the execution of a kernel. For example, according to the PCA, the size of DMA transfers, the number of DMA transfers, the number of live registers at context switches, the extent of memory accessible by each kernel, and the number of double precision instructions are all statically available metrics that should be good predictors of kernel performance.

In this example, we use the polynomial form of linear regression to determine a relationship between static program metrics and the total execution time of an application. Modeling M variables, each with an N-th order polynomial, requires at least $N \times M$ samples for an exact solution using the least squares method for linear regression. This limits the degree of our polynomial model in cases where only a few samples are available, which may be a concern for models that are built at runtime as a program is executing.

Though linear regression will generate a model for predicting the execution time of a given application on a particular CPU or GPU, it can generate predictions that are obviously not valid. For example, it is common for the model to predict short running kernels to have negative execution times, or predict that the execution time of a relatively more powerful processor is significantly

slower than another processor that is invariantly slower in all other cases. In order to account for these cases, we applied a technique typically used in image processing, Projections Onto Convex Sets (POCS) [17], to each prediction. POCS works by applying a series of transformations to data, each of which enforces some a priori constraint on the data that must be true in all cases. If each successive transformation causes another invariant property to be violated, the iterative application of each transformation is necessary to find a result that satisfies all of the a priori constraints. In this case, we impose the constraints that all execution times must be greater than 0, and that all predictions for a given application on a given architecture should be within two standard deviations of the mean of all execution times of the same application on other architectures.

In this study, we recorded the metrics described in Section 4.3.1 and execution times of 25 applications on seven different processors. We used this data to predict the execution times of new applications on the same processor and the same application on different processors. We found that the models are most accurate when predicting the same application on a different, but similar style of architecture. For example, predicting performance on a GeForce 8800GTX is most accurate when training data is acquired from GeForce 8600GS and GeForce 280GTX. Predicting the execution time of a new application on the same processor is also relatively accurate. However, models that attempt to, for example, predict GPU performance given CPU training data are wildly inaccurate. In these cases, it is necessary to apply separate models for each cluster of an application.

Application Modeling. Our first experiment attempts to build a model for the execution time of a the remaining 13 applications on an NVIDIA 280GTX GPU using 12 randomly selected applications for training. We chose to only include results from the 280GTX in this chapter because the models for the other architectures yielded similar results. Figure 27 compares the predicted execution time to the actual execution time for each of the 25 applications. Note that this model is intended to be used at runtime in a system that launches a large number of kernels, therefore it should be perfectly accurate for kernels that it has already encountered. This model is the most

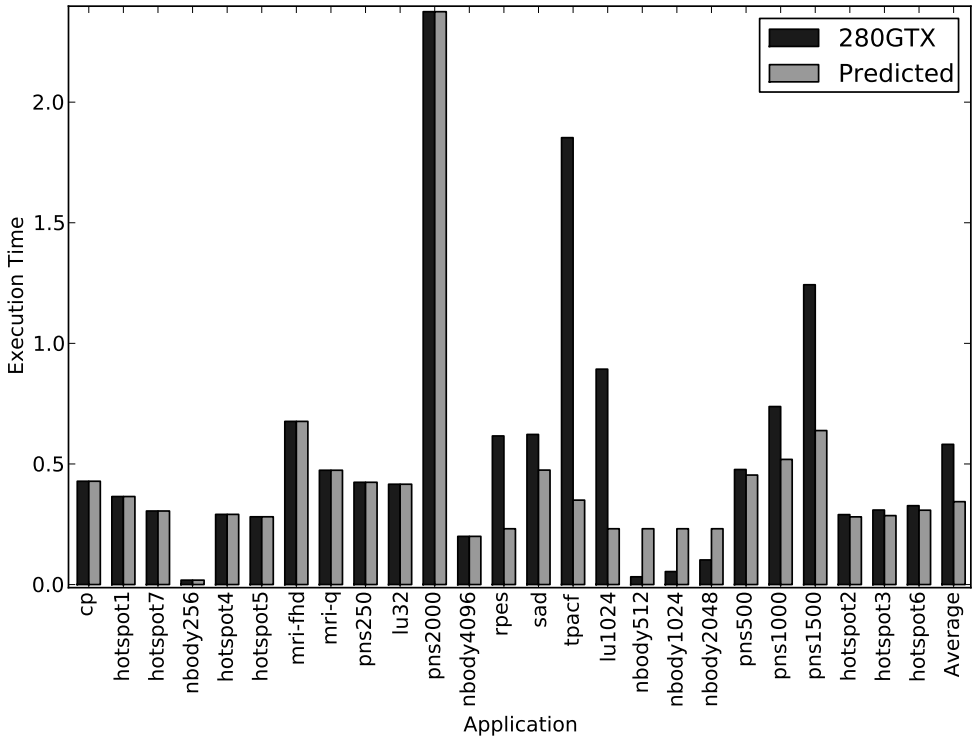


Figure 27: Predicted execution times for the 280GTX using only static data. The left 12 applications are used to train the model and the predictions are made for the rightmost 13 applications.

accurate for the hotspot, sad, and smaller sized pns applications where all predictions are within 80% of the actual execution time. It is relatively inaccurate for the *nbody*, *tpacf*, *rpes*, *lu*, and larger *pns* applications. In the worst case, the execution time of tpacf is predicted to be only 22% of the recorded time.

GPU Modeling. The next experiment uses the actual execution times of each application on the Geforce 280GTX, Geforce 8600GS, and Tesla C1060 to predict the execution time of the same applications on the Gefore 8800GTX. Figure 28 shows the predicted execution time as well as the measured execution time for each application. This model was the most accurate that we evaluated, the worst case being the *pns2000* application, for which to total execution time is predicted to be 3.9s and the actual execution time was 2.8s. In all other cases, the model underestimates the performance of the 8800GTX by between 16% and 1%. It is worthwhile to note that this model

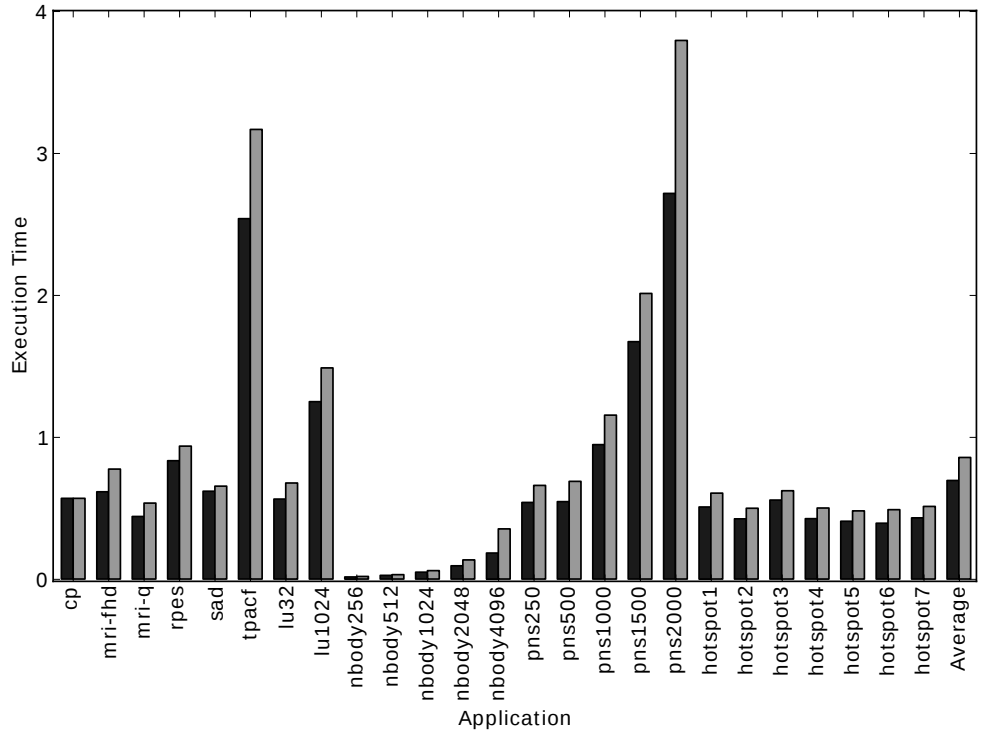


Figure 28: Predicted execution times for the 8800GTX using only static data and all other GPUs to train the model. Black indicates the measured time, and gray is the predicted time.

is able to predict the impact of increased problem size on the same application. For example, the execution time of each run of the *nbody* application is predicted to increase as the number of bodies simulated increases. The Parboil benchmark *tpacf* is relatively difficult to be predict by this model, and, in fact, the CPU and application models as well. This model always places the performance of the 8800GTX between that of the 8600GS and the Tesla C1060, which is also true for the actual execution times of all applications.

CPU Modeling. The third experiment uses training results from the Intel Nehalem and Intel Atom processors to predict the performance of the AMD Phenom processor. It should be immediately clear that moving to CPU platforms changes the relative speed of each application. For example, on all of the GPU processors, the performance of *sad* and *rpes* are relatively similar. However, on the CPUs, *sad* is nearly 25x faster than *rpes*. This reinforces the point that GPUs and

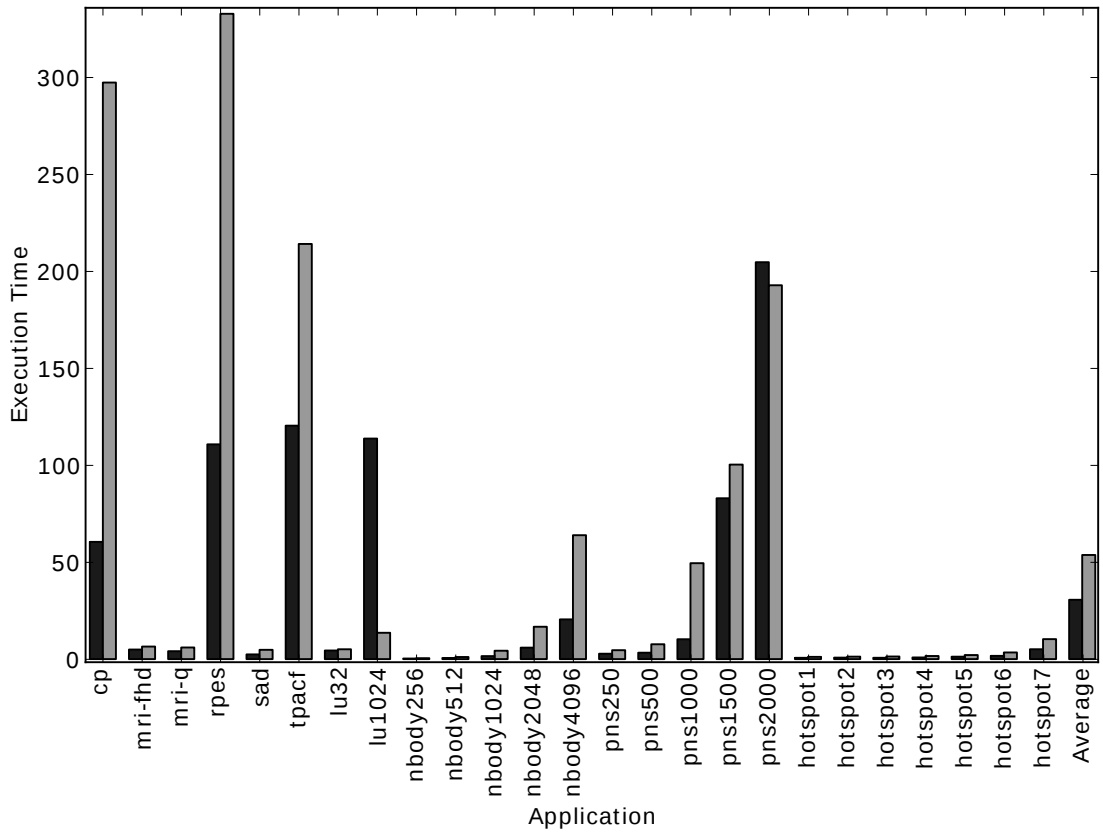


Figure 29: Predicted execution times for the AMD Phenom processor using the Atom and Nehalem chips for training.

CPUs are more efficient for certain classes of applications than others even when they are both starting with the same implementation of the program.

Compared to the GPU model, the CPU model is slightly less accurate as can be seen by comparing Figure 28 and Figure 29. The CPU model in Figure 29 is the most accurate for *hotspot*, the larger *pns* benchmarks, the *mri* benchmarks, and *nbody*. For those applications the predicted execution times fall with 80% of the measured execution times. The model is the least accurate for the *cp* application, which is predicted to take 62s and actually takes 294s to execute. It is interesting to note that this application only takes 54s on the Intel Nehalem processor, which is typically competitive with the AMD Phenom. Whatever machine characteristic causes this large discrepancy in performance is not captured in our set of machine metrics. It is possible that a more

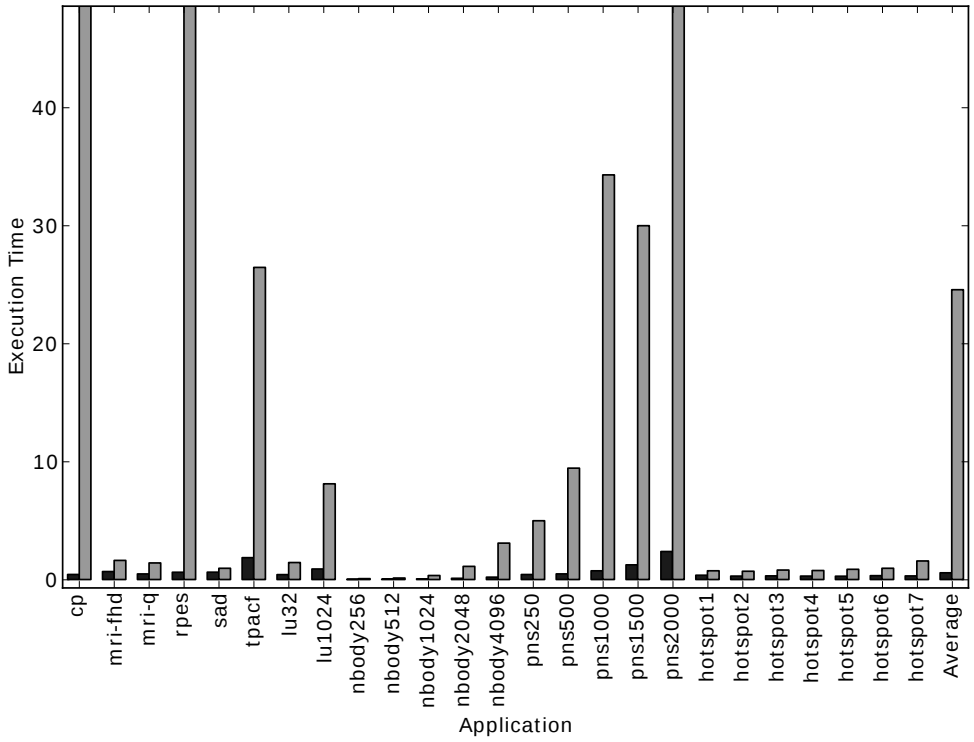


Figure 30: Predicted execution times for the 280GTX using all of the other processors for training. This is the least accurate model; it demonstrates the need for separate models for GPU and CPU architectures.

detailed model including more machine metrics would be able to capture that relationship.

GPU-CPU Modeling. The final experiment demonstrates a case in which our methodology fails to generate an accurate model. Figure 30 presents the predictions for a model for the 280GTX using results from all other processors for training. This model excessively overestimates the execution time of each application with only *cp*, *hotspot*, and *sad* being within 50% of the total execution time. As our cluster analysis shows, GPU and CPU style architectures have very different machine parameters and combining them in the same model significantly reduces the accuracy of the model. It motivates the need for a two stage modeling approach in which applications and processors are first classified into related categories with similar characteristics and then modeled separately.

4.4.5 Performance Model Validation

This set of experiments evaluates the capacity to *automate* the creation of performance models via the Eiger modeling framework. Automation completes the performance modeling infrastructure and enables a realization of a prototype like Eiger to be deployed in actual systems, to make decisions related to scheduling for heterogeneity and fast large-scale simulation. We evaluate Eiger via the following experiments.

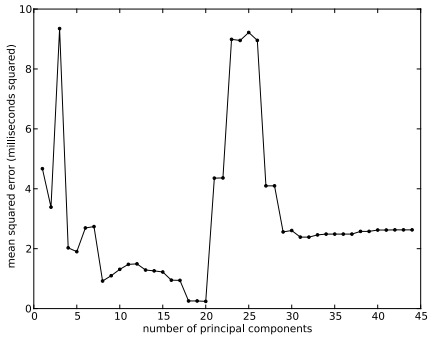


Figure 31: Error as dimensions are varied.

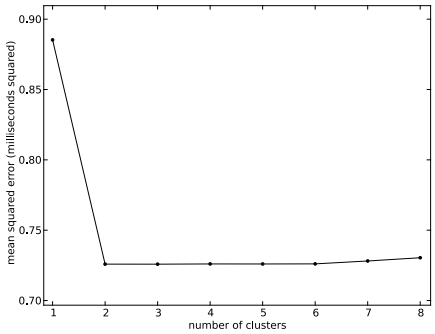


Figure 32: Error as number of clusters are varied.

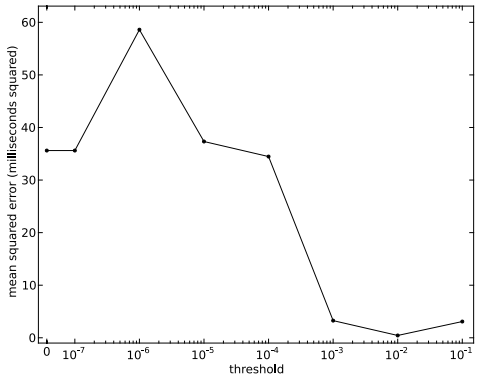


Figure 33: Error as convergence threshold varies.

Varying Dimensionality. This experiment reduces the number of dimensions retained after principal component analysis (PCA). Reducing dimensionality is a lossy compression technique, which

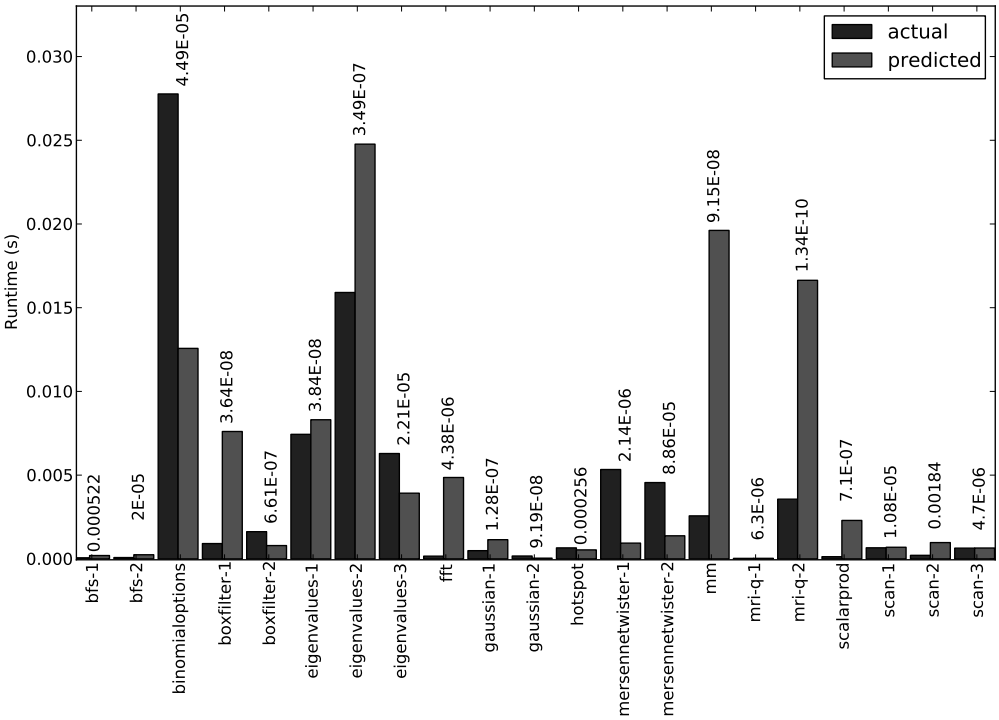


Figure 34: Predicted versus actual runtimes for each application for models trained from all other applications, annotated by mean squared error.

has the potential to alias important metrics which may have a great impact on the performance of the model. However, increasing the dimensionality of the input data has the potential to complicate the model, resulting in poor generalizability due to over fitting. As seen in Figure 31, there is indeed a minimum where the benefit of reduced dimensions on model complexity are balanced out by the loss of data.

Varying Cluster Count. This experiment increases the number of clusters to demonstrate the performance benefits of generating models from only like data points. Each data point in the experiment set is predicted by the model whose cluster's centroid is the closest. The results, shown in Figure 32, demonstrate how the segmentation into separate clusters for modeling can increase the quality of the models, although at a certain point the quality of the individual models decays due to the low number of data points in the cluster.

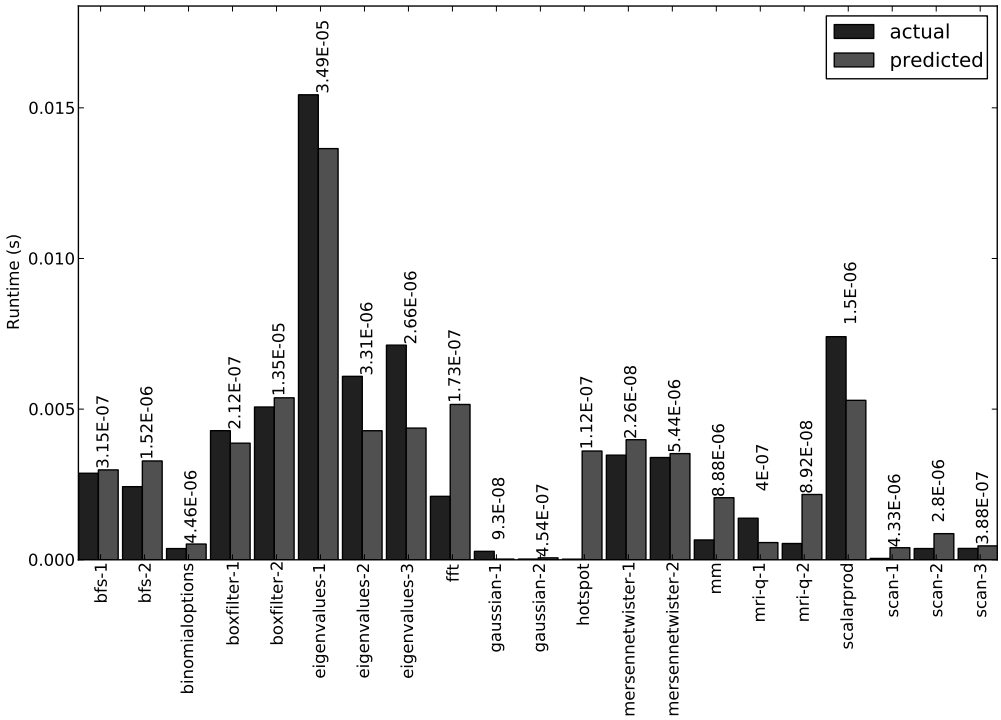


Figure 35: Predicted versus actual runtimes on the GTX 480 when model is trained from the GTX 560Ti and the Tesla C2070, annotated by mean squared error.

Varying New Function Threshold. This experiment reduces the threshold for adding new functions to the final model. As the threshold decreases, more functions that only marginally increase the quality of the regression are included. This increases the dependence on the training data and therefor the variance in the final model. The effect of varying this threshold is demonstrated in Figure 33.

Varying Applications. In this experiment each application is predicted using models created from all of the other applications. This experiment demonstrates how generalized the application metrics are; instead of relying upon previous runs of the same execution of the application to train the model, which may obscure some of the application characteristics (e.g. algorithmic complexity, communication patterns, etc.), other applications with potentially widely different algorithmic

implementations are used. Results are shown in Figure 34. Each application kernel is listed separately, indicated by the number concatenated to the end of the application name.

Varying Machine Parameters. In this experiment the GTX 480 is predicted by models trained on only the GTX 560Ti and the Tesla C2070. This experiment demonstrates how well hardware metrics can describe the performance of a given application. Results from this experiment are shown in Figure 35.

4.4.6 Discussion

The above experiments demonstrate the viability of automated model selection. The first experiment motivates PCA for dimensionality reduction, showing a clear reduction of error as the number of dimensions is increased. Increasing dimension count increases the complexity of the model though does indicate a trend of reduced model error to a point (20 dimensions) at which error increases substantially. This indicates a limit to performance model complexity via statistical methods, though this also depends on the number of trials and extent of metric variance considered. We also see a clear benefit to cluster analysis, a result that matches intution. Workloads with significantly differeing characteristics are unlikely to be good performance predictors of each other. Finally, these results show a trend of increasing model accuracy as the convergence threshold increases (and consequently, model complexity rises). This presents the designer with several tradeoffs in which tolerating increased model complexity and data acquisition can lead to greater accuracy. Online systems making performance predictions on the fly may benefit from simple models trained on few samples. Large-scale distributed simulations, on the other hand, are already achieving great reductions in workload through statistical performance modeling and may tolerate extensive training phases and complex performance models. This work demonstrates an automated model selection technique that enables fine-tuning of this tradeoff between complexity and accuracy.

4.5 Related Work

Performance prediction and modeling is a valuable tool to performance tuning, scheduling, and processor modeling.

Analytical GPU models. Analytical modeling examines the design of processors and systems and expresses the relationship between program inputs and machine configuration to several consumables. These consumables may be the number of operations, duration of time, or the amount of energy used during a computation. Hong et. al. [80] propose a predictive analytical performance model for GPUs. The main components of their model are memory parallelism among concurrent warps and computational parallelism. By tuning their model to machine parameters, static characteristics of applications, and regressions, their performance model predicts kernel runtimes with errors of 13% or less. Our approach, on the other hand, does not assume particular processor architecture or machine model and instead attempts to determine them based on measurable statistics that may change substantially as microarchitectures evolve.

GPU-Simulators. Like Ocelot, Barra [32] and GPGPU-Sim [9] provide micro-architecture simulation to CUDA programs. Similar to Ocelot, these simulators intercept GPU kernel invocations and execute them on a functional simulator. They are primarily concerned with reproducing the executions of kernels on actual GPUs, and characteristics derived from such simulations are influenced by it.

Scheduling for Heterogeneity. Rising levels of heterogeneity present system designers with opportunities to specialize processors and cores for particular tasks, devoting additional resources such as die area, energy, and execution priority to performance-critical applications. This complicates scheduling methodologies, as performance and efficiency may be greatly impacted by the mapping of applications to cores. Some predictive mechanism is needed to determine which mapping of workloads to cores is likely to optimize a given performance metric, including runtime, power, fairness, or other. Diamos et al. [50] show static partitioning of workload to core type

is insufficient as platforms change and *a priori* design decisions become obselete. Craeynest et al. [42] identify the problem of performance estimation for single-ISA heterogeneous processors and describe a technique for collecting runtime statistics such as ILP, IPC, and MLP, and estimating performance if core type were varied.

Statistical Simulation. Goswami et al. [65] perform detailed characterization studies of GPU applications using a detailed cycle-accurate simulator and identify redundancies in characterization metrics among kernels from different CUDA benchmark applications. The authors decompose kernel executions to provide recommendations for prioritizing the execution of benchmarks to exercise the most complete set of program behaviors in as few trial executions as possible.

Cook et al. [37] describe a Monte Carlo method for design space exploration and performance prediction using sampling and statistical methods. They sample several key hardware performance counters from real processors to generate histograms of microarchitectural events such as pipeline stalls, branch mispredictions, and cache misses. Once the probability distribution of stall events has been estimated, they are used to drive a generalized model of an out-of-order processo. Genbrugge et al. [62] describe a method for constructing a synthetic trace of a program execution bearing the same statistical properties as a complete execution but of much shorter overall length. They demonstrate a speedup in simulation time due to substantial reduction in size of the synthetic trace relative to a detailed trace.

4.6 Concluding Remarks

This chapter presents an emulation and translation infrastructure and its use in the characterization of GPU workloads. In particular, standard data analysis techniques are employed to characterize benchmarks, their relationships to machine and application parameters, and construct predictive models for choosing between CPU or GPU implementations of kernels based on Ocelot's translation infrastructure. A significant result of this study is that the methodology of principal components analysis, cluster analysis, and regression modeling is able to generate predictive models for

CPUs and GPUs, suggesting that there are certain characteristics that make an application more or less suitable for a given style of architecture. Unfortunately, while the regression method used in this study can generate an accurate model, it usually includes complex non-linear relationships that are difficult to draw any fundamental insights from.

This study exposes several non-intuitive relationships between application characteristics, for example, that applications with highly uniform control flow are more amenable to fine-grained synchronization and inter-thread communication. Discovering these relationships is becoming increasingly important as they can expose opportunities for architecture optimizations and influence the selection of well structured algorithms during application development.

This study demonstrates the Eiger performance modeling framework for automating the generation of statistical performance models of throughput-oriented workloads. Eiger provides an automated method in which designers may profile and characterize workloads, automatically construct performance models, and evaluate performance sensitivity to processor configurations. Our results show models based on as few as 5-7 principal components achieve fairly low mean squared error, but that adding principal components can increase squared error. We also verify cluster analysis has a strong impact on model accuracy due to significant differences in workload characteristics among benchmark suites. Finally, this empirical evaluation shows an automated statistical performance modeling framework such as Eiger provides an accurate approach to design space exploration of candidate microarchitectures when trained with sufficiently varied applications, data sets, and machine configuations.

Table 8: Metrics for each of the Parboil benchmark applications using the default input size.

Metric	CP	MRI_FHD	MRI-Q	PNS	RPES	SAD	TPACF
Static							
Extent_of_Memory	1112592	582630	517229	720002676	59883768	8948776	5009948
Context_switches	0	0	0	19	5	2	4
Live_Registers	0.00000	0.00000	0.00000	23.15000	4.60000	5.50000	14.50000
Total_Registers	20.000	18.500	17.000	35.000	27.000	21.000	22.000
DMAs	11	17	11	224	6	3	3
DMA_Size	1.5351e+05	4.6501e+04	6.7397e+04	7.1420e+01	1.1537e+07	2.9996e+06	1.6602e+06
Integer_arithmetic	3410	31080	5388	13102152	200196	1620	448
Integer_logical	0	12516	2136	2300592	111150	280	72
Integer_comparison	220	8582	1400	2244480	25560	142	68
Float_single	5280	17290	2908	617232	1279008	186	18
Float_double	0	840	0	0	0	0	0
Float_comparison	0	1050	180	0	38448	0	6
Memory_offchip	1760	2478	468	1094184	10530	66	14
Memory_onchip	770	1862	332	1178352	142434	198	64
Control_instr	660	11466	1968	3226440	186300	182	134
Parallelism_instr	0	0	0	533064	15030	12	8
Special_instr	880	0	0	0	43254	0	0
Othe_instrr	0	0	0	0	0	0	0
Instrumented							
Activity_Factor	100.000	100.000	100.000	97.200	63.850	95.400	80.510
Memory_Intensity	0.010000	0.060000	0.040000	4.640000	2.740000	5.880000	0.010000
Memory_Efficiency	49.200	49.600	49.600	65.600	73.100	47.700	48.100
Memory_Sharing	0.00000	0.00000	0.00000	51.50000	76.60000	2.90000	12.40000
SIMD Parallelism	128.000	292.570	320.000	248.880	40.580	70.280	206.110
MIMD Parallelism	2.5600e+02	1.1057e+02	9.7500e+01	1.7990e+01	6.4757e+04	5.9400e+02	1.5663e+02
Emulated							
Integer_arithmetic	2.2596e+08	3.7218e+07	2.2805e+07	5.2469e+10	2.1425e+10	1.5161e+07	1.3488e+09
Integer_logical	0	1.3738e+07	7.8520e+06	2.4229e+10	8.9300e+09	5.9380e+05	2.2153e+08
Integer_comparison	1.1267e+08	2.6135e+07	1.2572e+07	5.6280e+09	5.0089e+09	6.0093e+05	2.0857e+08
Float_single	2.9293e+09	2.1333e+08	1.1878e+08	2.1047e+10	4.1966e+10	6.0445e+06	6.0892e+07
Float_double	0	11010048	0	0	0	0	0
Float_comparison	0	1.3738e+07	7.8505e+06	0	1.2119e+09	0	1.4098e+08
Memory_offchip	4.5056e+05	3.8136e+04	1.0896e+04	5.3392e+09	6.8786e+08	1.9127e+05	2.5571e+05
Memory_onchip	4.5064e+08	1.3801e+07	6.3024e+06	4.8278e+08	1.4313e+10	4.4984e+06	3.2223e+08
Control_instr	1.1278e+08	4.5258e+07	2.6641e+07	5.7445e+09	1.5469e+10	8.2229e+05	8.4251e+08
Parallelism_instr	0	0	0	4.1073e+07	2.6514e+09	1.9008e+04	2.0161e+07
Special_instr	9.0112e+08	0	0	0	9.9957e+08	0	0
Other_instr	0	0	0	0	0	0	0

CHAPTER V

EXECUTION MODEL TRANSLATION

Execution model translation refers to mapping the structure of one particular computational model onto the logical resources of another. For example, transforming the explicitly parallel thread hierarchy of the PTX execution model - consisting of *kernels* launching grids of *cooperative thread arrays* - onto the compute resources of multicore CPUs. Valiant [151] describes execution models as bridging models, revealing important characteristics of the underlying physical realization of computation in an actionable manner to software such that algorithms may be designed to reduce important costs, including memory transfer, communication among concurrent processing elements, and computation within a processor. The critical elements of an execution model are the nature and scope of threads of execution, constraints to control and data flow, concurrency, communication domains, and isolation.

5.1 SPMD Execution on Vector Architectures

This thesis asserts that an explicitly parallel execution model may be dynamically compiled to processors exposing different types of parallelism. GPUs exhibit both coarse-grain and fine-grain parallelism. Multicore CPUs, on the other hand, expose coarse-grain parallelism among cores and hardware threads. Fine-grain parallelism is present in the form of instruction-level parallelism and vector parallelism via instruction set extensions. The techniques described herein may be leveraged by future architectures which achieve performance scaling through wide vector units and tight coupling between CPU and GPU cores on die. This decision has been validated by recently available and annnounced architectures from ARM, Intel, and AMD.

This chapter focuses on techniques for leveraging the PTX execution model as a portable data-parallel execution model and translating it onto the execution model and physical resources present in mainstream heterogeneous CPUs. Section 5.2.1 describes details in translating a throughput-oriented scalar instruction set to modern CPU instruction set architectures. In Section 5.2.2, we describe compilation techniques for reducing the amount of concurrency of the PTX thread hierarchy to relatively few coarse-grain hardware threads. Section 5.3 describes how up to 4x speedups are achievable by targeting specialized vector functional units through program transformations.

5.1.1 Source Execution Model

In [49], we have argued the PTX execution model is an instantiation of bulk-synchronous parallelism (BSP) described by Valiant [151] due to constrained synchronization domains and weak consistency guarantees of the memory hierarchy. BSP describes a bridging model which emphasizes fundamental costs in parallel computation, notably related to synchronization and communication. By elevating synchronization and communciation as fundamental elements of the execution model, the programmer may reason about communication overheads and achieve scalable implementations for massively parallel platforms. BSP decomposes parallel computations into supersteps composed of the following phases: (1) computation, (2) communication, and (3) synchronization.

This structure directly corresponds to the PTX execution model at two levels. At the highest level, each kernel launch may be considered a superstep in which computations (cooperative thread arrays) are invoked. These perform some computation, updating the global address space, then terminate. Writes become globally visible before the next kernel launch, and these correspond to synchronization and communication phases of the BSP model. At the CTA level, threads themselves may compute, synchronize, and communicate through shared memory. Unlike at the kernel level, threads may continue executing beyond synchronization steps thereby preserving a significant amount of live state without incurring off-chip data movement costs (not accounted for

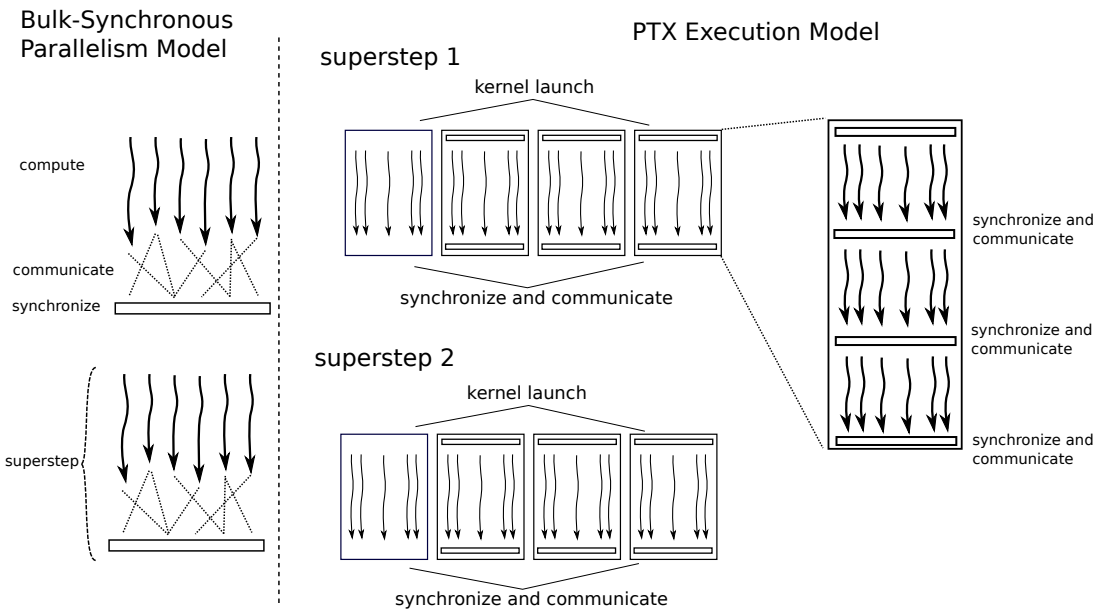


Figure 36: Bulk-Synchronous Parallelism mapped onto PTX execution model.

in the BSP model). The PTX thread hierarchy, abstract machine model, and relationship to bulk-synchronous parallelism are illustrated in Figure 36. Consequently, this work leverages PTX as the source execution model for scalable parallel program representations and explores techniques for expanding the set of processor architectures that may efficiently execute programs with this structure.

5.1.2 Target Machine Model

In the context of the PTX execution model, this work is concerned with mapping a collection of scalar *threads* from one or more *cooperative thread arrays* onto one or more vector functional units. A vector register file serves as the source and destination of vector operations. Vector load and store instructions operate between memory and this register file. Vector functional units are organized into multiple lanes where each lane implements a single arithmetic or logical operation. A single vector instruction controls all lanes. The width of vector unit will be referred to as the *warp size* for compatiblity with GPU computing terminology. Unlike SIMD processors from NVIDIA

and AMD, vector units/lanes cannot implement arbitrary control flow which both simplifies the implementation and limits their applicability. In particular vector units are distinguished by the following.

- vector loads and stores fetch contiguous sequences (vectors) of scalar data
- vector operators are applied within a lane; lanes may not be arbitrarily masked
- conditional select operators may choose between two values in each lane

Historically, this machine model has been made available to existing instruction set architectures via ISA extensions. Applications must explicitly map computations onto vectors and then express these computations using the exact operations available by the underlying ISA. This presents both forward and backward compatibility problems. New applications could not be run on previous generations of processors without an additional compatibility code path. As new processors are released with wider vector units with additional operators, existing applications would not be able to utilize new capabilities.

This machine model is currently implemented in the pipelines of numerous commodity processors presently available including Advanced Vector Extensions (AVX), Streaming SIMD Extensions (SSE), AltiVec, and ARM Neon. Moreover, the current trend is increasing vector widths from four in the case of SSE to eight in the case of the recently available AVX. Proposed processor architectures such as Intel's Knights Ferry [136] suggest this trend will continue with increasing vector widths.

5.2 Dynamic Compilation Framework

The proposed compilation model is wrapped by an API front-end for heterogeneous computing. This implementation supports the CUDA Runtime API. PTX modules are explicitly registered with the runtime which immediately parses and analyzes kernels within the modules. These are added

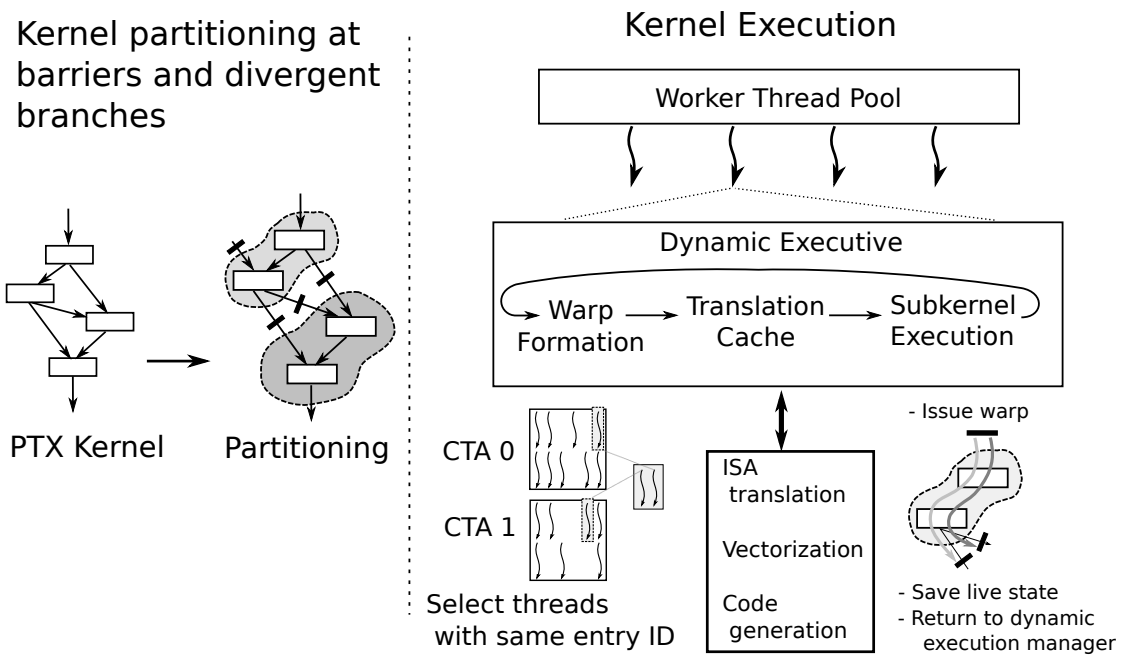


Figure 37: Dynamic compiler and execution manager framework for data-parallel kernels supporting vectorization.

to a global translation cache which lazily translates PTX kernels to LLVM and then vectorizes these translations for several warp sizes presented by the target machine model. Kernel launches, illustrated in Figure 37, spawn a set of hardware threads, each running a dynamic execution manager. The kernel's grid of CTAs is statically partitioned across the set of execution managers which concurrently serialize the execution of light-weight threads within the CTAs while respecting the semantics of the execution model. Execution managers form warps, or collections of PTX threads, waiting to execute the same block within the thread. The number of threads within the warp is used to query the global translation cache and obtain a native ISA binary. When threads reach a CTA-wide barrier or diverge, they yield via statically defined kernel exit points and control returns to the execution manager. This process iterates until all threads have terminated, and all worker threads reach a kernel-wide barrier at which point the kernel is finished.

Single-Instruction Multiple-Data (SIMD) processors have been classical instances of parallel

processors. By applying the same control logic (single instruction) to multiple arithmetic units, processor designers have saved die area. Moreover, SIMD magnifies the effective instruction issue bandwidth, as the same instruction drives many computations. This presents a savings in terms of both performance and energy efficiency. [70] has shown that as much as 30% of overal processor power is consumed by instruction fetch and decode units. This section presents a background of SIMD processors and presents a simplified machine model upon which subsequent discussions will build.

SIMD and vector ISA extensions to the x86 instruction set in the form of MultiMedia Extensions (MMX), Streaming SIMD Extensions (SSE), and Advanced Vector Extensions (AVX) introduced with Intel Sandybridge [83] and AMD Bulldozer [23] in 2011. These can be considered widely available heterogenous functional units, and current microarchitectural trends such as AVX, Intel's Larrabee [136], and the enormous growth of GPU computing research suggest that wide SIMD units are likely to be critical functional units in future processors.

5.2.1 Instruction Set Translation

Translating between radically different types of architectures requires mapping the execution and control structures of one model to another. Execution model translation is the main focus of this chapter, and mapping GPU execution model onto multicore heterogeneous CPUs is the main contribution. Implementing this translation requires the additional step of translating the source instruction set onto the target ISA. This section briefly describes the implementation used in GPU Ocelot for translating PTX to the x86-64 ISA. A detailed explanation of ISA translation from PTX to LLVM may be found in Section 7.3.3 as well as [67] and [49].

Efficient native execution on multicore requires a complete compiler backend including standard optimization passes, efficient code generation and selection, register allocation, and peephole optimization. This work leverages off-the-shelf modular components from the Low-Level Virtual Machine [105] project, particularly its optimization pass manager and code generator. This work

```
//
// CUDA
//
extern "C" __global__
void simple(int *A) {
  int i = threadIdx.x +
    blockIdx.x * blockDim.x;
  A[i] = i;
}
```

```
// PTX
.entry simple(
  .param .u64 _param_0) {

  .reg .s32   %r<6>;
  .reg .s64   %rl<5>;

  ld.param.u64 %rl1, [_param_0];
  cvta.to.global.u64 %rl2, %rl1;
  mov.u32     %r1, %ntid.x;
  mov.u32     %r2, %ctaid.x;
  mov.u32     %r3, %tid.x;
  mad.lo.s32  %r4, %r1, %r2, %r3;
  mul.wide.s32   %rl3, %r4, 4;
  add.s64    %rl4, %rl2, %rl3;
  st.global.u32   [%rl4], %r4;
  ret;
}
```

Figure 38: Compiling CUDA to PTX via nvcc. An LLVM translation of this kernel appears in Figure 39.

targets LLVM's intermediate representation, creating a set of LLVM instructions for each PTX instruction, and providing equivalent structures for other elements of PTX such as built-in special values, special functions, and rounding modes. PTX defines several instructions that affect how the execution model interacts with the target processor such as barrier synchronization, votes, and reductions. The execution model translation discussed in this thesis places special importance on these instructions in serializing threads.

As an example, Figure 38 presents the results of compiling a simple CUDA kernel to PTX. Figure 39 shows the translation of the same PTX kernel to the LLVM Intermediate Representation. In each case, the program is represented as a scalar function that is implicitly executed by a grid of threads. Built-in PTX variables such as `threadIdx` and `blockIdx` enable a thread to determine its identity within this grid and must be mapped onto logical thread indices during execution model translation. Classical scalar compiler optimizations may be applied to the LLVM representation, as long as constraints related to PTX semenatics are satisfied. This includes treating shared memory as volatile, explicit barriers synchronizing communication between threads, and requiring kernels to be deadlock free. See [119] for a detailed treatment of classical compiler optimizations.

```
//
// LLVM
//
define internal void @_kernel_simple_1_opt3_ws1(
  %LLVMContext* %__ctaContext) nounwind align 1 {

  %0 = getelementptr %LLVMContext* %__ctaContext, i64 0, i32 1, i32 0
  %blockDim.x.t0 = load i32* %0, align 4
  %1 = getelementptr %LLVMContext* %__ctaContext, i64 0, i32 2, i32 0
  %blockId.x.t0 = load i32* %1, align 4
  %argumentPtrPtr.t0 = getelementptr %LLVMContext* %__ctaContext, i64 0, i32 8
  %argumentPtr.t0 = load i8** %argumentPtrPtr.t0, align 8
  %ptrThreadCount = getelementptr %LLVMContext* %__ctaContext, i64 0, i32 12
  %rt9 = mul i32 %blockId.x.t0, %blockDim.x.t0

  %2 = bitcast i8* %argumentPtr.t0 to i64*
  %lsr.iv3 = bitcast %LLVMContext* %__ctaContext to i32*
  %threadId.x.t0 = load i32* %lsr.iv3, align 4
  %r0 = load i64* %2, align 8
  %r5 = add i32 %threadId.x.t0, %rt9
  %rt10 = sext i32 %r5 to i64
  %r6 = shl nsw i64 %rt10, 2
  %r7 = add i64 %r0, %r6
  %rt11 = inttoptr i64 %r7 to i32*
  store i32 %r5, i32* %rt11, align 4

  ret void
}
```

Figure 39: Translating the code in Figure 38 from the PTX instruction set to LLVM IR.

5.2.2 Scalar Thread Fusion

The PTX thread hierarchy emphasizes the creation of tens or hundreds of threads, thus maximizing the amount of expressed parallelism and enabling performance scaling as processors add more data paths. Multicore CPUs, even those featuring wide SIMD instruction set extensions, support the concurrent execution of few to tens of threads. Naively launching one kernel thread per PTX thread would result in frequent context switching and kernel-mode execution. Table 9 lists the MIMD and SIMD concurrency available in several commodity multicore CPUs.

Table 9: Concurrency of multicore CPUs with SIMD ISA extensions.

	MIMD	SIMD	Logical threads
Intel Core i7	8	8	64
AMD Fusion	16	8	128
Intel Many Integrated Core	16	16	256

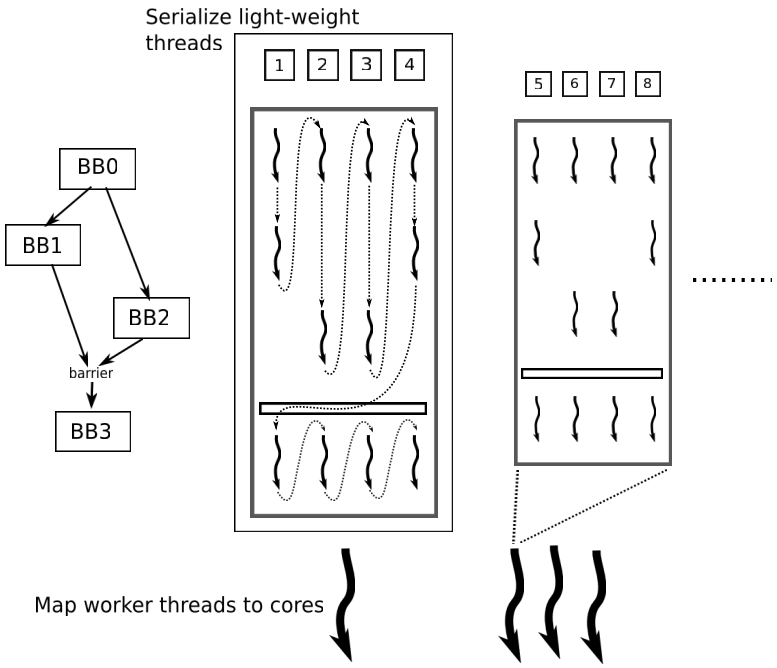


Figure 40: Thread fusion for efficient execution on multicore.

This work proposes fusing threads using compiler-inserted context switches effectively implementing a cooperative multithreading execution regime. The functional requirements of a traditional context switch require live thread state including register values, program counter, and stack pointer are saved to the thread's own stack, and the new thread's state be loaded into the register file. The C programming language and standard library provides *setjmp()* and *longjmp()* to save the entire register file and load a new thread's register file as needed. However, the entire architectural register file of 32-bit and 64-bit x86 processors is considerably larger than the average number of live values for most CUDA programs. Table 10 lists the state requirements for x86 and ARM instruction set architectures. The second column lists the average number of live values at context switch locations which require threads to store live state. The third column lists the size in bytes of these values. Results are presented for CUDA SDK, Parboil, and Rodinia benchmark suites, and are averaged over dynamic instruction traces for all kernels executed by each application. These results show compiler-inserted context switches that explicitly load and store values

may enable lower state requirements of CUDA workloads to be executed more efficiently. Moreover, this presents the opportunity for additional optimizations such as rematerializing values to further reduce state requirements. This optimization is explored in Section 6.

Table 10: Architectural register file size and average liveness of CUDA programs.

	Live Values	State size (bytes)
32-bit x86	16	64
x86-64	32	128
ARM	16	64
CUDA Applications	4.54	18.6

Liveness at Entry Points. This metric counts the average number of values restored per thread at entry points from the execution manager taken during the execution of each program. Runtime overhead at transitions between the execution manager and the kernel is proportional to the number of values restored. On average, 4.54 values are live per thread at each entry. Figure 41 illustrates this for each of the benchmark applications. Most applications with barriers have live state at yield points and require some context to be reloaded. On average, fewer values than architectural registers need to be restored indicating compiler-inserted context save and restore points may be at least as efficient as other types of cooperative threading libraries.

5.2.3 Impact of Scalar Thread Serialization

Serializing and executing threads fused in the manner described accomplishes the goal of reducing the amount of parallelism with low overhead. Context switches happen as infrequently as possible and store and restore only state that is provably live. However, this presents several implications on how instructions are reordered and have some impact on performance. Additionally, parallelism explicit in the program representation is discarded when threads are executed in series. In this section, we investigate the impact reordering memory accesses has on cache behavior and determine

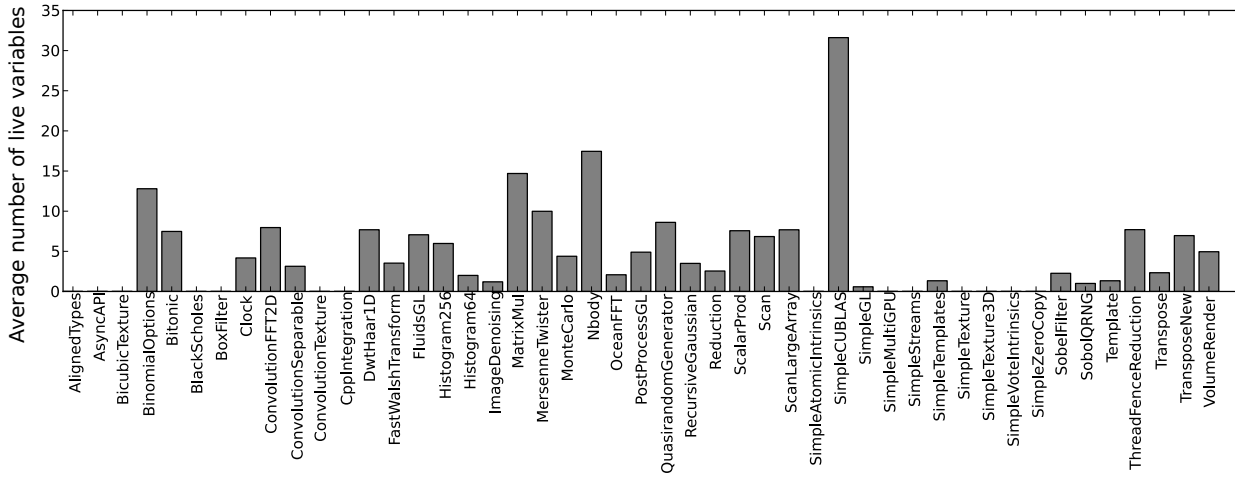


Figure 41: Average number of values loaded per thread on entry from the execution manager.

the extent of instruction-level parallelism exposed in the workload. All measurements are performed on a real-world platform [1] using the PAPI [20] platform-independent interface to hardware performance counters.

Memory Reordering. The memory efficiency metric described in Section 3.3.2 provides a measure of how much spatial locality PTX kernels are capable of exploiting. Lacking large caches, GPU architectures instead motivate programming models in which collections of threads access nearby lcoations in memory such that their accesses may be coalesced in time and issued in as few requests as possible, each fetching one cache line. This metric is defined and explained in greater detail in Section 3.3.2 and illustrated in Figure 16.

The first microbenchmark explores the impact of memory traversal patterns on memory bandwidth. This experiment is derived from prior work into optimal memory traversal patterns on GPUs, which indicates that accesses should be coalesced into multiples of the warp size to achieve maximum memory efficiency. When executing on a GPU, threads in the same warp would execute in lock-step, and accesses by from a group of threads to consecutive memory locations would map

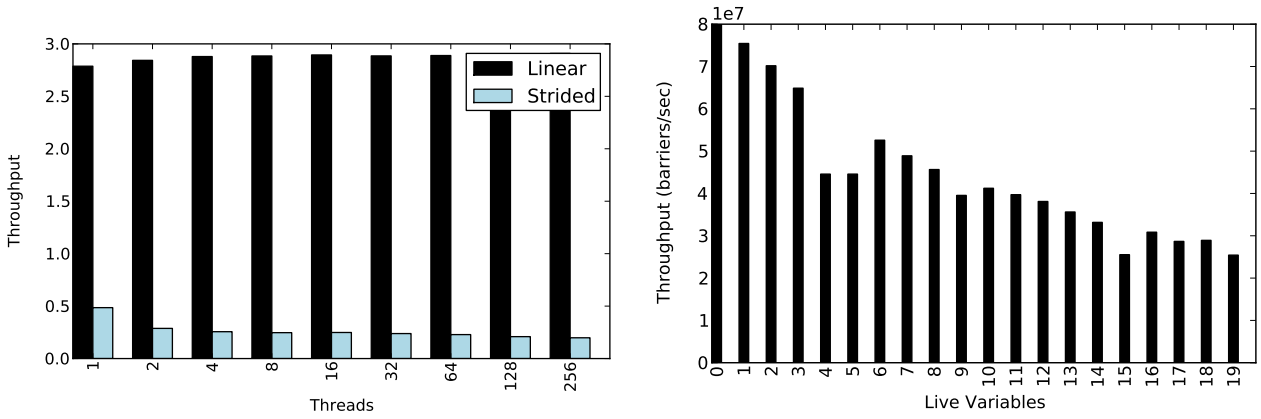
[1] Intel Sandybridge Core-i7, Ubuntu 12.04

to contiguous blocks of data. When translated to a CPU, threads are serialized by thread-fusion and coalesced accesses are transformed into strided accesses. Figure 42(a) shows the performance impact of this change. The linear access pattern represents partitioning a large array into equal contiguous segments and having each thread traverse a single segment linearly. The strided access pattern represents a pattern that would be coalesced on the GPU.

Insight: *Compiler Optimizations Impact Memory Traversal Patterns*. It is very significant that the strided access pattern is over 10x slower using the CPU backend when compared to the linear access pattern. This indicates that the optimal memory traversal pattern for a CPU is completely different than that for a GPU. PTX transformations, such as thread-fusion used in MUCDA [142], that change the memory traversal pattern of applications should be designed with this in mind.

Context Switch Overhead. This experiment explores the overhead of a context-switch when a thread hits a barrier. The test consists of an unrolled loop around a barrier, where several variables are initialized before the loop and stored to memory after the loop completes. This ensures that they are all alive across the barrier. In order to isolate the effect of barriers on a single thread, only one thread in one CTA is launched. The thread will hit the barrier, exit into the Ocelot thread scheduler, and be immediately scheduled again. Figure 42(b) shows the measured throughput, in terms of number of barriers processed per second. Note that the performance of a barrier decreases as the number of variables increases, indicating that a significant portion of a context-switch is involved in saving and loading a thread's state.

Multicore Scaling. The Parboil benchmarks were used as examples of real CUDA applications with a large number of CTAs and threads; previous work shows that the Parboil applications launch between 5 thousand and 4 billion threads per application [95]. Figure 43(a) shows the normalized execution time of each application using from 1 to 8 CPU worker threads. All of the applications scale well to two threads, but not necessarily beyond that. The CP benchmark is able to achieve better than a 4x speedup using 8 threads, indicating that it is probably compute bound and is able to benefit from SMT. Conversely, SAD slows down when the number of threads is increased beyond



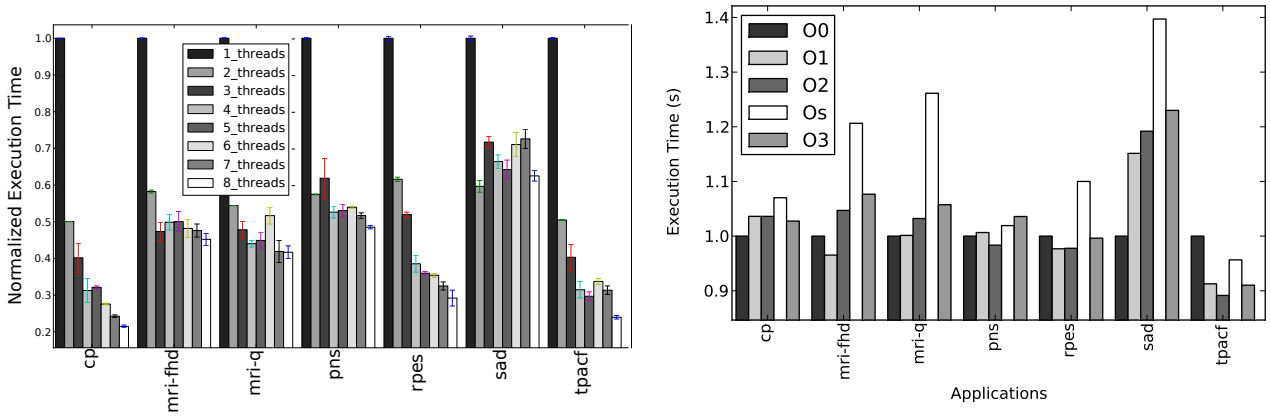
(a) Memory traversal.

(b) Barrier performance.

Figure 42: Thread fusion reorders memory accesses (a) and incurs overheads at barriers (b).

two. Previous work by Kerr et al. [95] have found PNS and SAD to be highly memory intensive, and likely to be constrained by a processor's off-chip memory bandwidth rather than its core count. These applications may be more suitable for GPU architectures, which focus on high bandwidth rather than low latency. These results motivate the need for a dynamic compiler like Ocelot that can direct applications to the most efficient architecture in a heterogeneous system.

Insight: *Variable CTA Execution Time*. Several of the applications in this paper demonstrate the importance of evenly distributing CTAs across cores in a CPU or GPU. These results suggest that work distribution schemes must simultaneously deal with two constraints that follow from locality among CTAs: 1) neighboring CTAs are likely to have similar execution times, and 2) neighboring CTAs are likely to access similar memory locations. In other words, mapping neighboring CTAs to the same processor core will improve memory locality, but lead to uneven work distributions. Conversely, random partitioning schemes will hurt memory locality, but even out work distributions. This motivates a work-stealing approach to distribute CTAs among hardware threads as well as to reduce the size of schedulable work items.



(a) Multicore scaling.

(b) Optimization scaling.

Figure 43: Performance impact of concurrency (a) and scalar optimizations (b).

5.3 Vectorizing Scalar Kernels

The scalar thread fusion technique described in the previous section has a straightforward implementation and achieves performance scaling on multicore CPUs. It does not however make any use of vector instruction sets. As described in the motivation of this thesis, SIMD functional units are a compact organization for parallel datapaths that are available in commodity CPUs and GPUs currently available. Modern multicore CPUs demonstrate up to a factor of 8x speedup of single-precision floating-point throughputs if SIMD instruction set extensions are utilized.

Figure 45 illustrates the transformation of a scalar instruction sequence into a vectorized form. Scalar load and store instructions are replicated, and the binary operator (floating-point multiply, in this case) is promoted to an element-wise vector operation. In the scalar kernel, two iterations would be required to execute this kernel over two threads. The vectorized kernel requires a single iteration for an equivalent execution and exhibits higher instruction-level parallelism.

5.3.1 Program Transformations

This work proposes *vectorization*, a program transformation mapping a kernel of data-parallel scalar threads onto a vector processor. This transformation produces a specialized form of a kernel

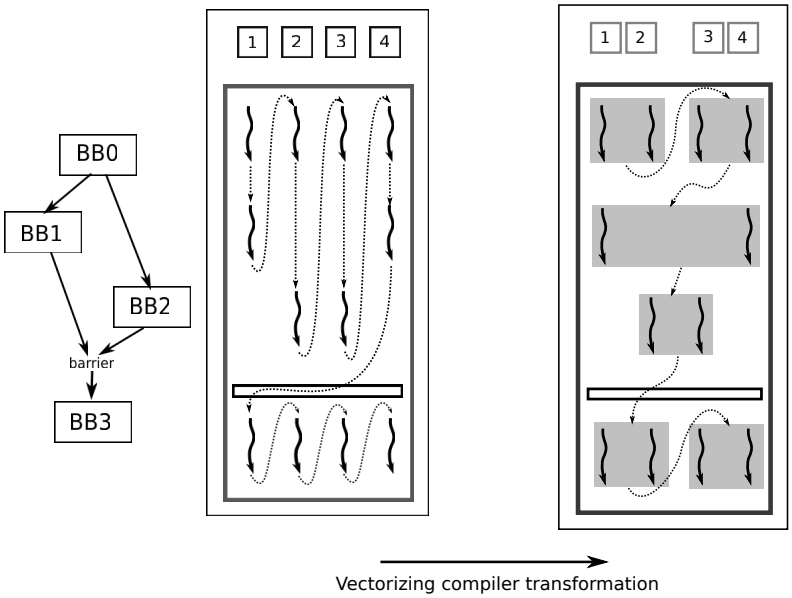


Figure 44: Vectorization logically fuses threads to exploit SIMD instruction set extensions.

by *replicating* scalar instructions and, where supported by the target ISA, *promoting* replicated instruction sets to vector operators. Execution of a single vectorized kernel is computationally equivalent to the serial execution of a scalar version of the kernel over a collection of threads where each thread is mapped to a lane within the vector functional unit, and the width of each vector operator is equivalent to the number of threads covered by this kernel's execution. Equivalence of executing scalar and vectorized kernels is illustrated in Figure 44.

Figure 45 illustrates the transformation of a scalar instruction sequence into a vectorized form. Scalar load and store instructions are replicated, and the binary operator (floating-point multiply, in this case) is promoted to an element-wise vector operation. In the scalar kernel, two iterations would be required to execute this kernel over two threads. The vectorized kernel requires a single iteration for an equivalent execution and exhibits higher instruction-level parallelism.

Vectorization may be implemented with Algorithm 5 whose input is a scalar kernel. Thread-local and CTA-local data members are accessed via a context object identifying the executing thread. This context object includes grid dimensions, block dimensions, block ID, thread ID, and

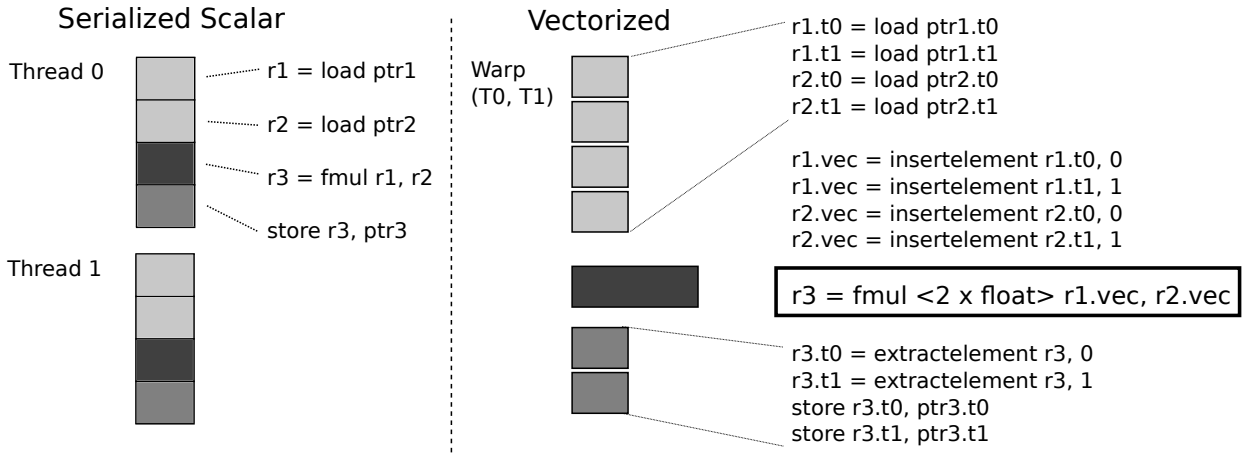


Figure 45: Serializing scalar threads executing the same basic block by interleaving static instructions and promoting arithmetic instructions to vector operators.

Input: Instruction i
Input: warp size ws
Output: Vectorized instruction
replicate i for each of ws threads
foreach *replicated instruction* **do**
  update thread ID operands
if i *is vectorizable* **then**
  replace ws instructions with single vector-typed instruction
memoize resulting instruction or bundle

Algorithm 5: `Vectorize`(i, w) replaces a scalar instruction a replicates set of instructions, one for each thread in the warp. This set may be promoted to a single vector instruction.

base pointers to the following address spaces: parameter, shared, and thread-local. The output is a vectorized kernel in which a single execution of each basic block is equivalent to executing that block by all of the threads in a warp. The input to the resultant vectorized kernel is an array of context objects, each describing a unique thread. This basic transformation does not consider divergence which is addressed by a subsequent transformation described in Section 5.3.2.

The vectorization pass is implemented as a transformation that replicates instructions while maintaining a mapping from scalar source instructions to the replicated set. Thread-local values

such as pointers to local memory and thread indices are loaded from a thread context object. Vectorized kernels receive an array of context objects constituting the warp, and accesses to context objects in vectorized kernels are modified to index the correct thread's context. Following replication, vectorizable instruction bundles are replaced by a single vector instruction with vectorized operands. Either the replicated instruction bundle or the vectorized instruction are memoized into the mapping. To vectorize the operands, they are either selected from the mapping, or they are recursively vectorized. The order in which instructions are vectorized affects recursion depth as well as performance of the compilation pass due to locality of the instruction objects and tabel lookups. The method adopted here is a breadth-first traversal of basic blocks composed with a linear scan of instructions within each basic block. This work vectorizes binary floating-point and integer operators as well as calls to transcendental functions for which both LLVM and the compilation target, the x86-64 ISA with AVX, have built-in support.

Non-vectorizable Instructions. CPU instruction set architectures do not typically support vector forms of all instructions. Loads and stores, for instance, do not support scatter and gather with vectors of pointers. Rather, many ISAs such as SSE and AVX enforce loading of contiguous data from a single base address. Significant performance may be lost if this value is not aligned to super-word boundaries. This approach groups loads and stores in a class of instructions which may be not vectorized and are instead replicated for each thread. The values produced are explicitly packed into vectors when a non-vectorizable instruction produces a value used by a vectorized instruction, and explicitly unpacked when a non-vectorizable instruction uses a vectorized operand. A subsequent dead-code elimination pass removes unused instructions. Conservative handling of load and store instructions as non-vectorizable enables this technique to accommodate misaligned accesses and accesses to random locations, two cases that would not perform well or are not supported by SSE.

Explicitly repacking scalar values into vectors presents some overhead, though the extent of additional data movement instructions emitted depends on the actual kernel being compiled and

on the quality of the backend code generator. In the particular case of memory instructions, we envision divergence analysis [41] and affine analysis [33] to identify opportunities in which multiple threads are guaranteed to access contiguous data. In these instances, arbitrary loads may be replaced with vector loads.

Sequences of interleaved replicated instructions exhibit instruction level parallelism that is at least as high as warp size. This comes at the cost of increasing the live ranges of values which places pressure on register usage. Moreover, transformations within LLVM's code generator attempt to subvert explicit instruction interleavings in order to reduce live ranges while discarding ILP. This required modifying LLVM to select an existing code generator that maintains the instruction schedules of source LLVM modules.

Implicit Synchronization. Guo, et al. [69] identiy idioms related to implicit synchronization among the threads in a warp when executing on SIMD processors. As an optimization, programmers rely on the hardware executing the threads of a warp in lock-step and omit barriers when threads in the same warp exchange data through shared memory. Omitting barrier instructions saves several cycles by not issuing the instruction, however such programs are not portable across processors with different warp sizes (from AMD to NVIDIA GPUs, for example). Moreover, it is not always possible for a compiler to address implicit synchronization, as not all threads in the warp may reach the implicitly synchronized code. The compiler cannot not insert a warp-wide barrier without risking incorrect behavior in the case when some but not all threads reach the implied barrier. To the best of our knowledge the technique proposed in [69] is not capable of handling such a case. The work described in this paper assumes the programmer does not require warp synchronous execution and yields undefined behavior for such kernels.

5.3.2 Divergent Control Flow

The set of threads mapped to a vectorized kernel must necessarily take the same control paths. An execution of a kernel is *convergent* if all threads follow the same path; execution is *divergent* if

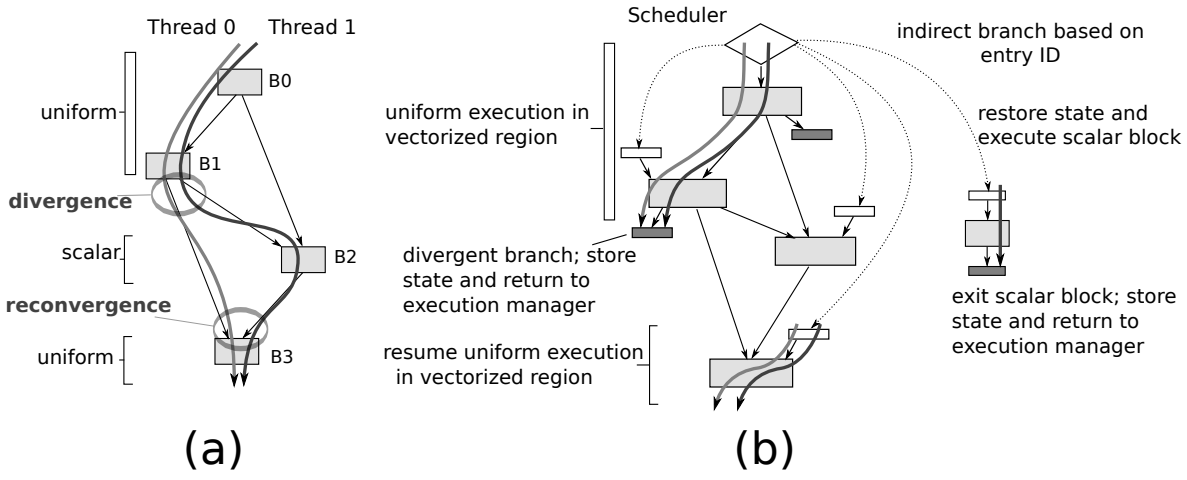


Figure 46: (a) Control-flow graph executed by two threads diverge at B1 and reconverge at B3. (b) Executing a kernel with divergent control flow through a vectorized and a scalar specialization of the kernel.

threads evaluate conditional control-flow instructions differently. Some kernels may be statically proven to be entirely convergent, and presumably some kernels contain potentially divergent paths that are never taken by common datasets. Characterization studies [95] indicate most real-world CUDA programs experience some form of divergence which must be efficiently tolerated. Figure 46 (a) shows a sample control flow graph with two threads executing B0 and B1 then diverging at the the branch terminating block B1. Thread 0 branches to B3 while thread 1 falls through to B2. A single execution of a vectorized basic block is equivalent to both threads executing the scalar form, therefore some mechanism must be present to avoid executing B2 for thread 0.

This work proposes *yield on diverge*, a software-only approach which checks branch conditions at runtime. Figure 46 (b) illustrates the execution of a kernel with divergence control flow. An execution manager collects a set of ready threads waiting to execute the same basic block. The execution manager then selects a vectorized kernel whose warp size is equal to the size of the collection of threads, and control enters the vectorized kernel. A scheduler block performs an indirect branch based on the identity of the actual entry point, and control resumes execution

within vectorized basic blocks.

Conditional branches are modified with additional instructions to detect divergent branches. On divergence, threads yield to an execution manager which inserts threads into a *ready* queue and forms a new warp. Execution of a vectorized block is logically equivalent to each thread within the warp executing that block, and consequently warps may only be formed of threads waiting to enter the same block. Yields to the execution manager are analogous to a context switch. Barrier synchronizations are handled like divergent branches except threads are inserted into a *waiting* queue within the execution manager.

Algorithm 6 describes how vectorized kernels are transformed to accommodate control-flow divergence. This applies the `Vectorize`(i, ws) function described in Algorithm 5 to vectorized instructions. Conditional branches terminating basic blocks are transformed by summing the branch predicates from each thread. If the sum is zero, all threads jump to the fall-through target (the branch was uniformly not taken). If the sum is equal to warp size, all threads jump to the branch target (uniformly taken). Otherwise, control enters an exit handler which performs the divergent yield. Successors to divergent branches are inserted into a list of possible entry points which are then used to construct a scheduler block at the beginning of the kernel.

Transitions from the execution manager to the kernel are accomplished via a compiler-inserted scheduler block which acts like a trampoline. A basic block inserted into the kernel contains a large switch statement conditioned on the warp's entry ID. These integer-valued IDs select basic blocks that are the successors to divergent branches (or barrier synchronizations) identified in Algorithm 6. For each entry point, an entry handler block is inserted to restore the warp's live state from local memory. Its terminator instruction jumps to the vectorized entry block. Algorithm 7 describes how a scheduler block is constructed.

Exit handling code inserted by Algorithm 8 into exit blocks performs yields to the execution manager. At yield points such as divergent branches and barriers, control passes from a vectorized block to an exit block. The exit block first spills all live values to thread-local memory for each

```
Input: Warp size ws
Input: Scalar kernel to be vectorized
Output: Vectorized function supporting control-flow
begin
    entrySet := {} exitSet := {}
    foreach basic block b in kernel do
        foreach non-control instruction i in b do
            Vectorize(i, ws)
        if b ends in conditional branch then
            insert empty basic block exit_b to function
            insert instruction: sum( predicates )
            replace the conditional branch with:
            switch sum( predicates ) do
                case 0
                    jump to fall-through successor
                case ws
                    jump to branch successor
                otherwise
                    jump to exit_b
            add exit_b to {exitSet}
            add successors(b) to {entrySet}
    CreateScheduler({entrySet})
    CreateExits({exitSet})
```

Algorithm 6: Inserts detection and handling code into kernel.

thread. Then, a conditional select operator stores a constant-valued integer identifying the branch target block for each thread which is then written to that thread's resume point field. Divergent threads will evaluate this select instruction differently and write different entry IDs to their resume point fields. Finally, a value indicating the disposition of the kernel exit is written to the warp's resume status field. The execution manager, described in Section 5.4.2, updates its pool of ready thread contexts according to the resume status type and chooses a new warp by collecting threads with the same resume point.

This work considers three classes of kernel yields: divergent branches, CTA-wide barriers, and thread termination. When yielding on barriers, the execution manager places context objects in a

Input: $\{entrySet\}$
create empty basic block `scheduler`
insert `switch` statement into `scheduler` with default target of entry block to function
foreach b *in* $entrySet$ **do**
    create empty basic block $\texttt{entry}_b$
    insert *load* instructions into $\texttt{entry}_b$ for all live-in values at block b
    insert *jump* to block b
    add to switch statement in `scheduler`:
    case *(*b*)*
        jump to $\texttt{entry}_b$

Algorithm 7: `CreateScheduler`($\{entrySet\}$) creates a scheduler block and inserts code to restore live state.

Input: $\{exitSet\}$
create local variable $resumeEntryId$
create local variable $resumeStatus$
foreach $\texttt{exit}_b$ *in* $exitSet$ **do**
    insert *store* instructions into $\texttt{exit}_b$ for all live-out values at block b
    insert $resumeEntryId \leftarrow$ ***select(***$predicate$*, {** ***branchTarget, fallThrough*** *})***
    insert $resumeStatus \leftarrow$ { `Thread_branch`, `Thread_barrier`, `Thread_exit` }
    insert `return`

Algorithm 8: `CreateExits`($\{exitSet\}$) stores live-out state at divergence sites, inserts a conditional select operator to specify the target entry point, specifies a status indicating why the warp has returned to the execution manager, and exits.

wait queue to avoid rescheduling them until all threads in the CTA have reached the barrier. On termination, the context object is discarded. This work does not implement function calls, mainly due to their relatively new introduction to programming model on which this work was evaluated. These may be potential sources of divergence also, either during conditional call instructions or indirect calls when the target is non-uniform. The approach described here may be extended to function calls via the introduction of a thread-local call stack, replacing call targets with kernel entry IDs, and by always yielding on function calls. This remains for future work.

Figure 47 illustrates entry and exit handlers in greater detail. Block B1 has been vectorized for warp of size 2 and exists within the shaded region shown in the figure. Block B1_entry has been added to the kernel and provides a control path from an external scheduler into the vectorized

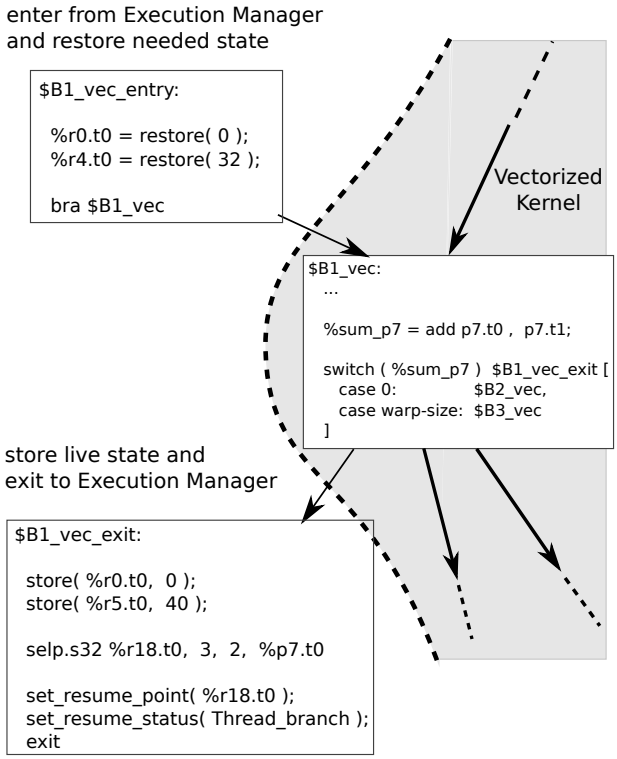


Figure 47: Divergent branch entry and exit handlers for a vectorized kernel. The conditional branch in the vectorized block B1_vec has been replaced by explicit checks. On divergence, threads yield by exiting via B1_vec_exit.

region that loads live values from thread-local memory. A conditional branch instruction terminating B2 has been replaced with a switch statement whose conditional is the sum of all branch conditions within the warp. Its default successor is the exit handler, and two other successors are vectorized blocks within the kernel.

This technique requires a scalar specialization and a specialization for some maximum vector width. Additionally, implementations may produce specializations for narrower vector widths. The implementation for this work assumes each kernel has been specialized for warp sizes of 1 thread, 2 threads, and 4 threads corresponding to available vector processing hardware in the target processor. Entry and exit points have been added to divergence and reconvergence sites to restore live variables from thread local memory and enter the kernel. A scheduler block performs an

indirect jump to the entry point selected by the warp's entry ID.

5.4 Implementation

The vectorization transformation described in the previous section was implemented as a device backend to GPU Ocelot [48], a dynamic compilation framework for GPU computing. GPU Ocelot translates PTX code to LLVM's IR and utilizes its extensive analysis, optimization and code generation facilities [49]. Our implementation described in this section extends GPU Ocelot's multicore CPU backend with the addition of a dynamic execution manager, dynamic translation cache, and the vectorization program transformation.

5.4.1 Dynamic Translation Cache

The translation cache is the module responsible for producing native ISA binaries of each kernel by translating from PTX to LLVM, applying program transformations, and JIT compiling to the native ISA of the target CPU. Exhibiting the external semantics of a code cache, it may be queried by execution managers running in the worker threads by specifying an entry point ID and warp size. Before translation to LLVM, a PTX to PTX transformation replaces non-branch predicated instructions with select and splits basic blocks at barriers. Entry and exit handlers are inserted with procedures to store and restore live values as well as update thread status and next entry points on kernel exits. The process of translating PTX to LLVM has been described in detail in [49]. This work leverages many of these techniques to translate scalar PTX kernels into LLVM representations but applies the unique approach to execution model transformations described in Section 5.3.

When kernels are launched, execution managers query the translation cache for particular warp sizes. These initiate translation from PTX to a scalar LLVM representation and subsequent vectorizing for the requested warp size. Potentially, the translation cache could be modified to support querying for additional specialization parameters beyond warp size such as optimization level or

particular kernel argument values. Techniques for specializing kernel regions are described in Chapter 6. Following translation and vectorization, the translation cache applies existing LLVM transformation passes including traditional compiler optimizations such as basic block fusion and common subexpression elimination. Finally, LLVM's code generator performs JIT compilation to yield a native ISA form of the vectorized kernel which is inserted into the cache to future requests from execution managers.

5.4.2 Dynamic Execution Manager

Each worker thread instantiates an execution manager which orchestrates the execution of all PTX threads from this set of CTAs while respecting CTA-wide barrier semantics (Figure 37). The execution manager contains a data structure of thread context objects, manages per-CTA memory structures such as shared memory and a block of contiguous memory partitioned into per-thread local memory. It implements warp formation and a thread scheduler. Prior to each kernel entry, the execution manager may select any thread not waiting at a barrier for execution. The current algorithm selects a ready thread via a round-robbin scheduler then attempts to construct the largest warp possible from other ready threads with the same entry point. The execution manager then calls the kernel and passes the warp of thread contexts. When threads yield to the execution manager, the warp's resume status indicates whether thread context objects should be terminated, returned to the ready pool, or added to their parent CTA's barrier pool.

Execution managers block while contending for lock on the dynamic translation cache. Compilation which is performed in the parent worker thread of the querying execution manager, so multiple worker threads querying for the same unavailable translation would be stalled. A possible optimization to the execution manager might give scheduling priority to warps for which translations exist to avoid stalling while the dynamic translation cache is actively compiling a previously requested translation. At this time, we only perform translations on kernel granularities so the benefits of concurrent execution and translation are less apparent. This is the subject of ongoing

work but is orthogonal to vectorization.

5.5 Experimental Results

This section presents the results from an evaluation of the described extensions to Ocelot-2.0.1464 compiled with LLVM 3.0. Evaluations were conducted on a desktop workstation running Ubuntu 11.04 x86-64 and using over 40 benchmark applications chosen from the CUDA Software Development Kit and the Parboil Benchmark Suite [81]. The evaluation system contains an Intel Sandybridge (i7-2600) CPU. Sandybridge supports SSE 4.2 and AVX. The proposed techniques for targeting vector functional units are expected to utilize AVX, but current lack of support for AVX in LLVM's code generator made such an evaluation infeasible as of this time. Moreover, this work is expected to apply to future architectures such as Intel's Knight's Ferry [136] equipped with 16-lane vector units. However, lack of simulation tools and a backend code generator for this ISA prevent an evaluation on this platform at this time.

The first set of experiments investigates speedups for idealized cases showcasing the benefits of vectorization and thread fusion. The second set of experiments measure performance improvements for real-world applications and provide statistics about application behaviors recorded by the execution manager. These statistical behaviors justify some design decisions and provide insights into sources of speedup and future optimizations. The third set of experiments evaluates the effectiveness of several proposed optimziations enabled by this dynamic compilation framework. This set of evaluations is intended to capture the performance gains possible using a portable data-parallel kernel representation that runs on GPUs and CPUs and is not necessarily intended to be competative with hand-tuned kernel implementations.

Throughput. This microbenchmark attempts to achieve peak theoretical throughput of floating-point units by replicating a sequence of interleaved, independent instructions. As described by Volkov [153], pipeline latency may be hidden given a sufficiently large number of threads. Increasing threads results in increased pressure on the register file, but the benchmark's relatively

small number of live values and non-overlapping ranges is easily to accommodated. Multicore CPUs are more heavily pipelined with issue latency of four cycles in the case of SandyBridge's SSE floating-point simple arithmetic unit [84].

Table 11: Peak floating-point throughput.

Warp size	1	2	4	8
GFLOP/s	25.0	47.9	97.1	37.0

Table 11 illustrates sustained floating-point throughput for increasing vector widths for a compute-bound kernel running on the test platform. Floating-point throughput is expressed in single-precision GFLOP/s on a machine whose peak floating-point throughput is estimated to be 108 GFLOP/s. The benchmark itself consists of back-to-back floating point multiply and adds within a heavily unrolled loop launched over 576 threads. Warps of 4 threads achieve 97.1 GFLOP/s on the target machine, or 90% of peak. Scalar threads saturate the scalar FPU issue ports and achieve 25.0 GFLOP/s. Exceeding the vector width of the target processor requires the code generator to emit multiple vector operators in series which increases register pressure and extends the live ranges of values. Consequently, executing the above benchmark with a warp size of 8 threads while targeting SSE results in degraded performance.

5.5.1 Performance Gains

Speedup. The principle benefit of vectorization is the efficient utilization of vector functional units for applications that exhibit divergent control flow. This set of evaluations captures runtimes of CUDA kernels from the CUDA 2.2 SDK and Parboil application suites for a maximum warp size equal to the machine vector width of 4 threads. Speedups relative to a baseline of scalar execution are presented in Figure 48. The baseline translator and thread scheduler is identical to what was presented in [49].

Average speedup is 1.45x. Speedup varies from approximately 1.0x in the case of applications

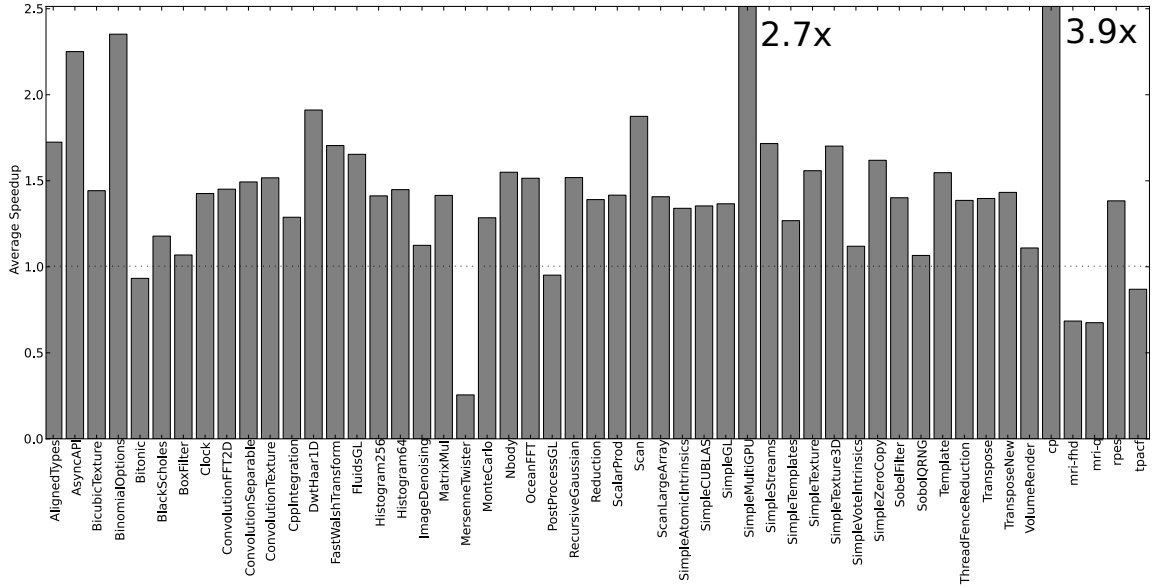


Figure 48: Speedup of benchmark applications.

such as *BoxFilter*, *ScalarProd*, and *SobolQRNG*. These applications have memory-bound kernels but perform frequent synchronizations such that threads maintain high locality even without vectorization. Other applications that are more compute bound with fewer synchronizations achieve higher speedups. *BinomialOptions* achieves 2.25x speedup over the baseline, and the Parboil application *cp* achieves 3.9x speedup. Both applications have very uniform control flow properties and unrolled loops. Other applications such as *MersenneTwister*, *mri-fhd*, and *mri-q* run slower with dynamic warp formation. We believe this is due to control-flow irregularity. Threads with uncorrelated control-flow properties may diverge at every branch unless maximum warp size is limited. This observation motivates future work to detect cases when diverging branches are so frequent that scalar execution is optimal.

Average Warp Size. Figure 49 illustrates the average warp size of each kernel for the applications executed in Figure 48. This metric expresses the fraction of kernels of warp sizes 1, 2, and 4, where 4 is the maximum warp size. The results indicate that most kernel entries from the

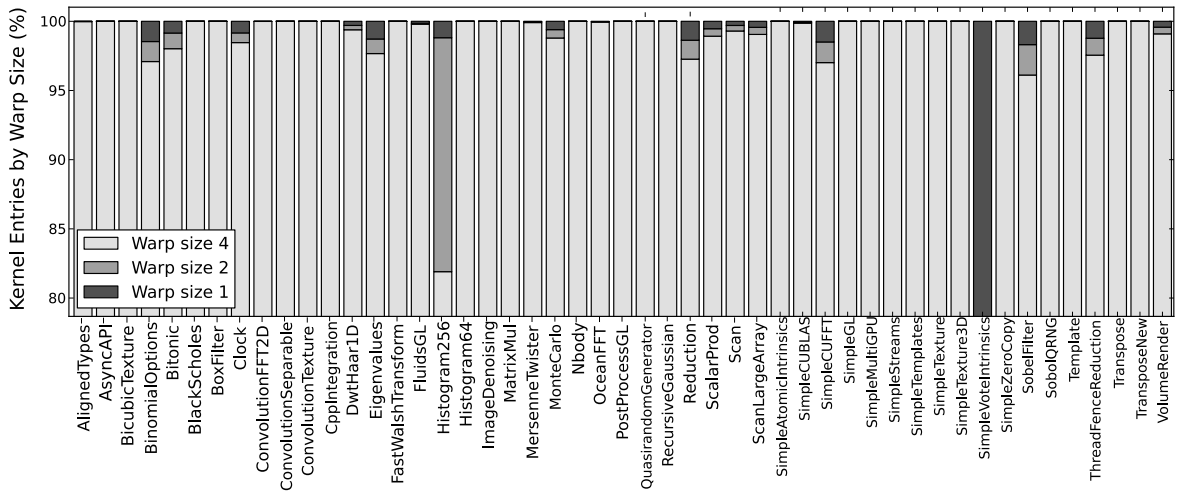


Figure 49: Average warp size of executed kernels with maximum warp size of 4 threads.

execution manager have warp size of 4 for every application except *SimpleVoteIntrinsics* which only forms of warps of at most two theads. These results also show that many applications are not entirely convergent which justfies the design decision to tolerate divergence and use dynamic warp formation to maximize available warp size. Finally, convergence does not entirely capture performance properties. *BinomialOptions*, for instance, achieves among the highest speedups.. This indicates that a dynamic warp formation strategy is very effective in improving utilization, but an implementation may be penalized by frequent kernel exits to the execution manager.

Yield Overheads. Figure 50 illustrates the fraction of CPU clock cycles spent in different phases of the execution of kernels in an application. For many applications, time spent in the Execution Manager (EM) is extensive. This includes testing barriers, inserting thread contexts into warps, and updating thread status after the execution of a kernel. Yield points that store and restore live state on transitions to the execution manager present a small overhead relative to cycles spent actually executing the kernel. Applications such as *MersenneTwister*, *Nbody*, and *CP* achieve both

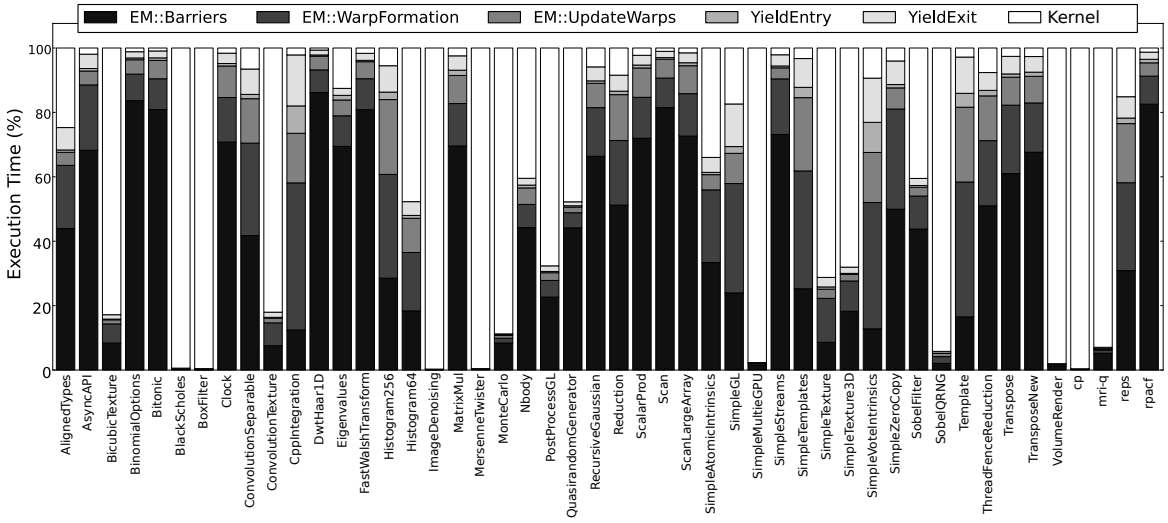


Figure 50: Fraction of cycles in execution manager (EM), yields to and from the EM, and executing kernel.

high speedup, and nearly all execution time is spent within the vectorized kernel. Synchronization-intensive applications such as *BinomialOptions* and *MatrixMul* spend more time within the execution manager and have limited speedup, even with little divergence. As vectorization reduces the execution time of the kernel, the relative percentage of time spent in the execution manager increases. These results suggest improving efficiency of the execution manager is key to further increases in performance even for applications with highly efficient compute-bound kernels.

5.6 Optimizations

The vectorization transformation exposes explicit parallelism within the intermediate kernel representation which may then contribute to several optimizations. This section describes three optimizations that make explicit use of thread-level parallelism within the instruction stream. **Thread-invariant elimination** constrains the warp formation algorithm to utilize consecutive threads and uses data-flow analysis to identify *thread-invariant* expressions. These are expressions with no

data-dependencies on thread-ID within the same CTA and therefore compute the same value. Serializing their computation in time is redundant and may be eliminated. **Affine analysis** identifies expressions computing the sum of some thread-invariant base value added to thread ID multiplied by a constant. Prevalant examples are memory addresses which may be vectorized if conservative analysis proves adjacent threads access consecutive and aligned memory locations.

5.6.1 Thread Invariant Expression Elimination

Scalar threads following the same paths through a kernel may compute identical results in some expressions. These expressions have data dependencies to CTA-wide invariants such as kernel arguments, block and grid dimensions, and shared constants. For example, many CUDA kernels compute the expression *blockDim.x * gridIdx.x*, such as when determining a thread's global index in a kernel grid. *Thread-invariant* expressions are redundant across a warp, and their elimination is expected to improve performance when threads are serialized. The approach to vectorization described in this chapter enables classical compiler optimizations - common subexpression elimination - to identify thread-invariant expressions and eliminate them.

This experiment constrains warps to consist of consecutively indexed threads, a mapping defined *a priori* and termed *static warp formation*. Following vectorization, thread ID values are replaced with constant expressions relative to the warp's base thread. For example, *thread 0* loads its thread ID from a context object, but *thread 1* computes it from *thread 0*'s ID. Expressions in each thread, which are not true dependencies of thread ID, are subsequently marked as thread-invariant. Standard common subexpression elimination optimizations downstream of vectorization eliminate redundant thread-invariant expressions via a conservative analysis.

Collange *et al.* [33] show an average of 15 % of PTX destination operands are reported as thread-invariant averaged over CUDA SDK applications. This work's approach to vectorization with static warp formation was able to reduce LLVM instruction counts by 9.5 % on average for a warp size of 2 threads. LLVM's optimization pass also reduces vector instructions to scalar

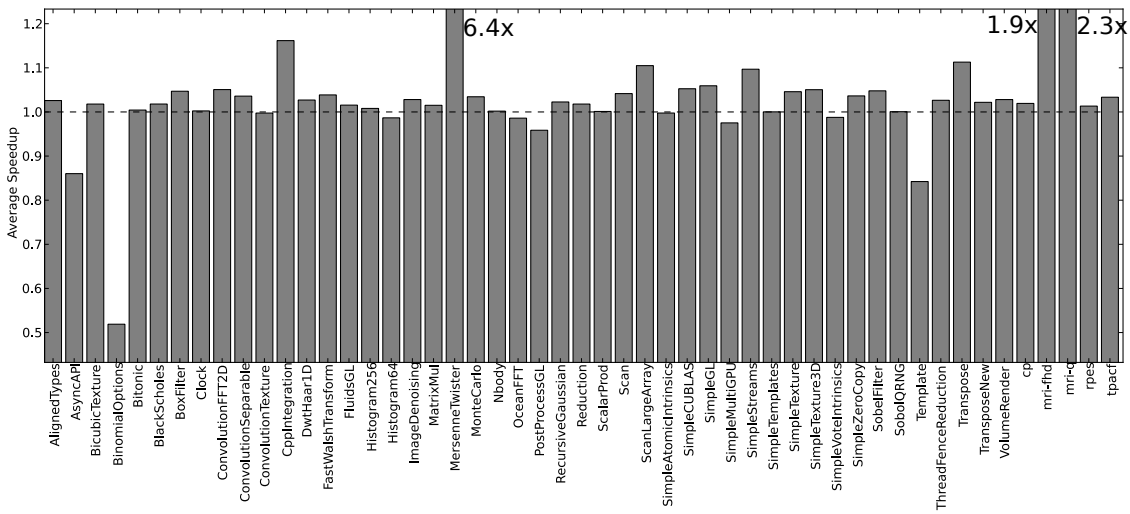


Figure 51: Speedup of static warp formation with thread-invariant elimination over dynamic warp formation.

instructions when lanes other than the base lane are redundant. For a warp size of 4 threads, 11.5% of instructions were eliminated. Larger warps imply a large fraction of thread-invariant instructions.

Speedup with Thread-Invariant Elimination. This experiment constrains warps to consist of consecutively indexed threads from the same CTA and applies thread-invariant elimination. Performance normalized to vectorization with dynamic warp formation is plotted in Figure 51. Average speedup is 11.3%, yet some applications achieve considerably higher performance with static warp formation than with dynamic warp formation. *MersenneTwister* experienced a 4.9x slowdown with dynamic warp formation, but static warp formation and thread invariance achieved a 1.30x speedup over completely scalar execution. The boost in performance is likely due to constrained warp formation in the presence of irregular control flow behavior.

5.6.2 Future Work

Affine Analysis. Execution within vectorized kernel is logically equivalent to the execution of multiple scalar threads, yet SIMD functional units and increased memory locality may be taken

advantage of. *Affine analysis* is an additional optimization which statically determines which expressions, if any, are linear combination of some uniform value and thread ID. For example, each thread may compute an address from a base value passed as a kernel parameter added to the product of thread ID and size of the data type to be written. Sampaio et al. [134] describe a static compiler analysis using gated static-single assignment form for detecting thread invariance and affine expressions in PTX kernels.

In particular, affine analysis enables identifying load and store operations in which threads of a warp access consecutive locations. Target instruction sets cannot vectorize arbitrary scatter and gather instructions, and so this implementation treats them as a sequence of scalar accesses. Proving accesses are consecutive enables promotion to a single vector load or store directly to vector registers offering a reduction in dynamic instruction counts. Moreover, this enables a further reduction in address arithmetic, as only a base address need be computed rather than an address for each participating thread. Listing 5.1 illustrates a vectorized instruction sequence in which a store instruction is provably affine in its destination address and may therefore be vectorized.

```
%rt2 = bitcast i8* %argumentPtr.t0 to i64*
%r0 = load i64* %rt2, align 8
%r5.vec = fadd <4 x float> %r3.vec, %r3.vec
%r6 = shl nsw i64 %threadId.x.t0, 2
%r7 = add i64 %r0, %r6
%vec.rt6 = inttoptr i64 %r7 to <4 x float>*
store <4 x float> %r5.vec,
  <4 x float>* %vec.rt6, align 16
```

Listing 5.1: Affine analysis successfully vectorizes affine store.

For this work, we implemented an analysis pass that enumerates data dependencies for all values used as addresses in load and store instructions. Value definitions and uses are then recursively traversed and terminated when one of several conditions are reached. If a value depends strictly on load instructions from constant memory, shared memory, argument memory, or global variables, it may be considered thread-invariant across the entire CTA and is marked accordingly. Multiply instructions are marked as affine when one operand is thread-invariant and the other is equivalent to thread ID. Some analysis is required to identify equivalence thread ID, as these values may

be obscured by integer casting as well as restores from local memory. Affine analysis applies data-flow analysis to identifying structural properties of expressions in an effort to identify those expressions whose results may be expressed as a baseline value added to the product of thread ID and a constant:

$$v = baseline + c * threadID$$

Affine expressions may be replaced by a single computation of the baseline value and then successive interleaved computations for each thread. This optimization is useful when the baseline value is computationally intensive, and eliminating redundant copies for large warps enables a considerable savings in instructions. It is also beneficial for certain classes of instructions which may be vectorized for affine structures, memory accesses in particular. Many CPU architectures enable vector memory accesses in which a single base address loads a vector of consecutive data elements. PTX, on the other hand, enables each thread to compute an arbitrary pointer value, so naively mapping threads to lanes of a vector memory access is insufficient. Affine analysis identifies memory accesses which are vectorizable by proving threads access consecutive locations of the correct size.

Rematerialization [19, 25] is a feature of register allocation algorithms and implementations that determines whether the cost of spilling a register truely outweighs the cost of recomputing its value. If the space requirements or spill costs are greater than the cost to recompute the value, and if the operands needed to recompute the value are available, subsequent uses should be preceeded by code to recompute its value rather than restoring it from the spilled memory region.

Rematerialization can be applied to an explicitly parallel source execution model to alleviate several problems. First, executing a large number of threads implies a significantly greater amount of live state. The execution model requires certain concurrency guarantees such as at barriers,

and lock-step execution of vectorized code implies at least one warp of threads are active when the kernel is being executed. Thus, one spilled value over a warp of eight threads can exceed the capacity of one L1 cache line. As presented in Figure 41, barriers and divergent control-flow typically average 4.54 double-word sized values per thread for each context switch.

5.7 Related Work

Karrenberg [90] and Shin [137] present approaches to vectorization that focus on conditional select operators. These works replace conditional control-flow with conditional data-flow and rely on predication in combination with control-flow graph restructuring transformations to accommodate divergence. Predication is a light-weight technique for disabling divergent or terminated threads along some control paths but reduces SIMD utilization. Instructions predicated off which cannot be vectorized incur additional penalties, as they occupy pipeline stages yet their results are discarded. Their evaluation includes several optimizations that were not implemented for this work such as coalescing of affine vector loads and stores.

Stratton *et al.* [142] propose several approaches to translate the PTX execution model for efficient execution on multicore CPUs. Stratton describes a source-to-source translator that inserts nested thread loops into the control structures of a CUDA kernel's abstract syntax tree. Live values spanning multiple thread loops are expanded into arrays indexed by thread ID. Scalar threads are entirely serialized, and memory accesses are dramatically reordered across threads. Diamos *et al.* [49] describe the translator from PTX to LLVM on which this work was based. The multicore CPU backend to GPU Ocelot serializes scalar threads similarly to [144] and [142], though these implementations do not make use of vectorization.

In [141], Steffen *et al.* present a hardware mechanism for terminating kernels that have executed divergent branches and spawning continuations that execute after a grouping phase chooses threads waiting to execute the same branch target. This technique incurrs overheads for all branches regardless of uniformity and does not immediately support CTA-wide synchronization barriers.

Launching continuations on control-flow divergence through specialized hardware support is similar to what the dynamic execution manager of this work performs using software.

G-Streamline [159] controls the incidence of thread divergence by re-mapping tasks to threads. While such a re-mapping is trivial in implementations of the proposed technique, this work does not investigate scheduling nor task distribution heuristics to maximize control-flow utilization. Rather, it assumes that in any mapping, divergence is possible and requires a context switch betweeen specializations for different warp sizes.

Other approaches to portable vectorization such as Liquid SIMD [29] proposed by Clark *et al.* encode vectorizable operations as sequences of annotated scalar operators that are promoted to vector types by a dynamic compilation environment at runtime. This provides portability in terms of vector widths without incurring significant translation overhead. However, Liquid SIMD is applicable to program representations that have already been vectorized, perhaps with a technique such as proposed in this work. It does not approach the problem of vectorizing collections of scalar threads with correlated control flow. Barik [11] *et al.* present an algorithm for efficiently and automatically vectorizing scalar code by forming short vectors from independent instructions, using horizontal vector operators, and by algebraic simplification. Like other classical approaches to vectorization, the initial representation is not a collection of data-parallel threads but instead focuses on vectorizing scalar threads. The set of optimizations proposed is complimentary to dynamic vectorization presented here and might be used after vectorization to improve the quality of generated code.

5.8 Conclusions

This research shows explicitly data-parallel kernels can be compiled for efficient execution on modern multicore CPUs leveraging vector and SIMD functional units while tolerating control-flow divergence. We present a program transformation for specializing a kernel representation for various vector widths and propose a method for accommodating divergent control flow instructions

via a light-weight virtual context switch implemented by compiler-inserted handling blocks. A dynamic execution manager orchestrates the execution of collections of threads by forming warps from a pool of ready threads with identical entry locations. An implementation of this technique is evaluated within GPU Ocelot, a research compilation framework for heterogeneous platforms. We apply dynamic vectorization to real-world workloads from existing GPU compute applications.

Microbenchmarks demonstrate near-peak computational throughput on GPUs and, with dynamic vectorization, peak throughput on an Intel Sandybridge CPU with vector ISA extensions. This technique is expected to scale across multiple vector widths and is not coupled to features of particular instruction set extensions. Consequently, it is applicable to other processor architectures with vector accelerator units such as PowerPC and ARM. Moreover, this technique does not require hardware support for divergence and provides dynamic compilation support for deploying data-parallel kernels on systems composed of both GPUs and multicore CPUs.

CHAPTER VI

REGION-BASED COMPILATION AND SCHEDULING

6.1 Introduction

Compilation for heterogeneous platforms presents challenges in balancing portability with performance. Highly tuned implementations perform well on targeted processors but may be inefficient on vastly different architectures or fail to run altogether, perhaps due to ISA incompatibilities or constraints in the execution model. Dynamic compilation systems closely tied to the execution of heterogeneous workloads facilitate portability by enabling a single representation of software to be compiled and executed on a variety of processors assuming the initial representation is suitably high level and that runtime compilation overheads can be mitigated. Elements of the source program representation may be difficult to implement practically in all target architectures and may require costly and inefficient emulation for complete coverage. Frequently, this causes conservative compiler analysis to fail to apply powerful optimizations that take advantage of difficult-to-use hardware [113].

Specializing regions for specific optimizations overcomes constraints in static optimization by enabling a runtime to select the fastest running form of a region of code assuming initial preconditions are true, yet this risks code explosion if many forms are compiled. Approaches toward dynamic specialization using runtime-detected values have been coupled with efforts to reduce JIT compilation time and code size expansion by maintaining a single form of the kernel and updating the binary representation few the kernel via an execution manager [100]. Coupled with value profiling for function parameters [99], significant performance improvements are possible. Speculative execution [7, 149] enables optimizing beyond what is statically provable and utilizes

specialized hardware support to roll-back state in the event that preconditions for certain optimizations have been violated. This incurs a runtime penalty during mis-speculation as well as possible code expansion to handle unoptimized cases.

This chapter describes a novel technique to mitigate the effects of code expansion by reducing the size of compilation regions while simultaneously exploiting highly correlated behaviors among threads within Single-Instruction Multiple-Thread (SIMT) workloads. This work shows region-based compilation improves JIT compilation costs, better exploits instruction-level parallelism (ILP) and memory divergence for data-parallel kernels, and enables specialization for aggressive optimizations for multiple backend targets. We propose partitioning data-parallel kernels into regions - *subkernels* - and specialize subkernels via dynamic compilation based on predicted and observed execution characteristics. Execution managers executing in concurrently makes detailed thread scheduling decisions based on control paths taken and thread context state when compiler-inserted yield points are reached, and light-weight thread loops efficiently fuse logical thread execution. This work defines several kernel partitioning heuristics intended to optimized for computationally intensive inner loops, and revisits program transformations utilizing SIMD functional units with additional optimizations. This chapter makes the following contributions:

- We propose the use of subkernels for the region-based dynamic compilation of data parallel kernels
  - We present partitioning heuristics for subkernel formation
  - We describe compiler and runtime architecture for managing subkernel execution
- Thread scheduling and code generation optimizations for subkernel-based compilation targeting multi-core CPUs
- Implementation and performance evaluation over a range of CUDA applications

6.2 Background and Motivation

Dynamic compilation is required by design decisions behind the OpenCL API. OpenCL requires high-level source distribution of kernels without a standardized lower-level intermediate representation, although recent efforts such as AMD's HSAIL [121] have been initiated to propose such a standard. CUDA is coupled to its low-level virtual instruction set, Parallel Thread eXecution (PTX) [123], but presents no binary interface directly to the hardware. Targeting devices other than commodity GPUs requires extensive compilation and translation infrastructure which too must support dynamic compilation. Thus, dynamic compilation is inescapable, and overheads may be particularly expensive when executed on low-power embedded processors or when invoked on short-running kernels. Chapter 5 proposes specialization for vectorization and potentially incurs significant code expansion.

This work extends the execution model translation techniques described in Chapter 5 by first partitioning kernels according to their control structures prior to dynamic compilation. This compilation flow, shown in Figure 52, starts with a (1) highly parallel, architecture-independent specification of application components (kernels), (2) partitions kernels into *subkernels* for separate compilation, (3) performs architecture-specific, reductions of parallelism that fit the resources available in target processors (thread fusion), (4) generates code for the target processors, (5) executes the application utilizing all of the available hardware resources. The novel step of partitioning for separate compilation and scheduling enables a significant reduction in code expansion, particularly when compiling specialized forms of each region for vectorization. Smaller regions for thread scheduling enable improved memory access patterns in spite of more frequent context switches.

6.3 Subkernel Partitioning

Subkernel partitioning enables (1) smaller compilation units and (2) fine-grain thread scheduling. Figure 53 describes how the program representation of kernels are partitioned and how threads are

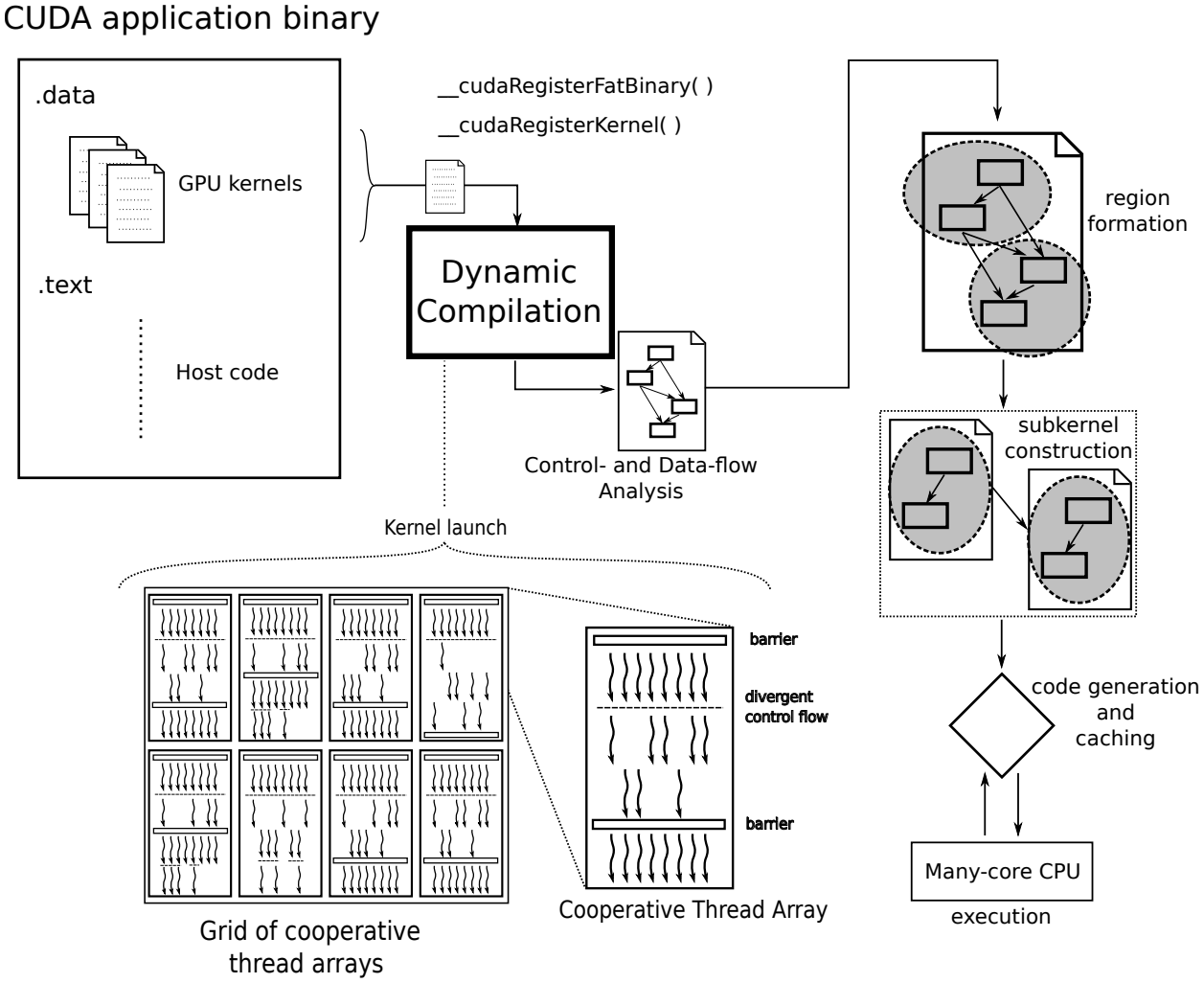


Figure 52: Dynamic compilation of kernel-oriented programming model for heterogeneous architectures.

scheduled to execute the resulting partitions. Statically, the kernel is partitioned into disjoint *subkernels*, and a directed graph is constructed to describe control edges between subkernels. At runtime, launching a kernel allocates thread blocks (CTAs) to a pool of worker threads. Each worker thread instantiates an Execution Manager which owns a set of thread context objects corresponding to the logical threads within the CTA. On demand, the EM groups threads by the subkernel they are ready to execute and demands a reference to compiled subkernel from a centralized code cache. On a cache miss, the worker thread blocks while the code cache dynamically compiles and

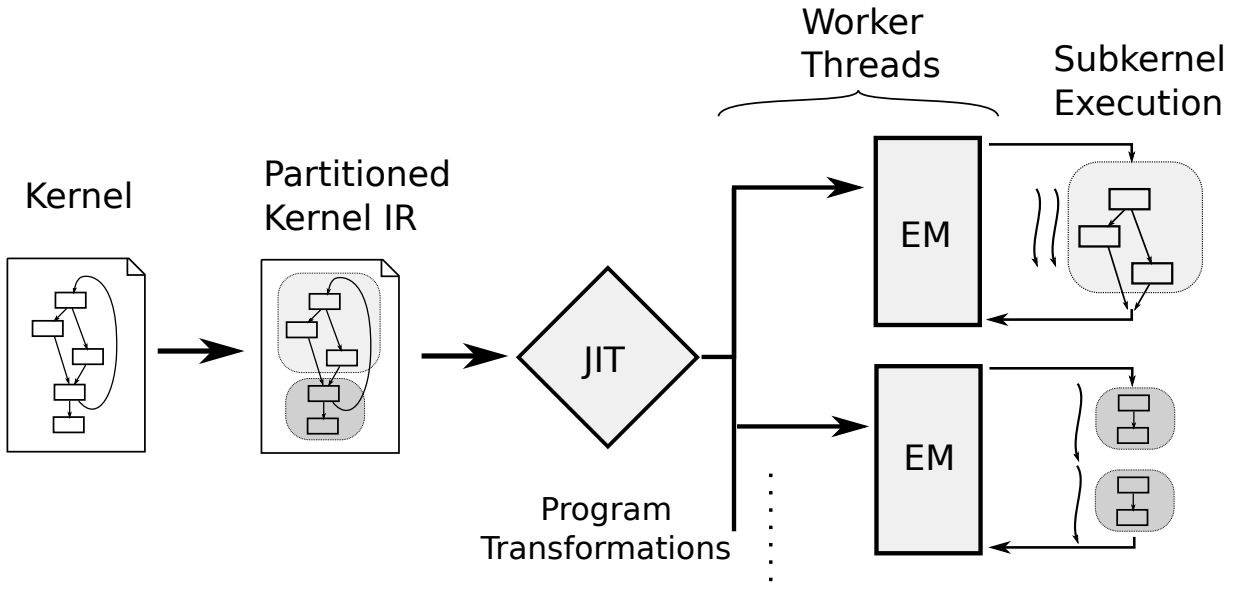


Figure 53: Subkernel partitioning, JIT compilation, and execution by virtual machines.

optimizes the requested subkernel.

Subkernels may be compiled lazily thereby avoiding high startup costs associated with just-in-time compilation of entire kernels. Regions containing entirely dead code are never compiled, and compilation of subsequent subkernels may be overlapped with kernel execution. Moreover, subkernels present narrowly scoped regions that enable aggressive optimizations and specialization, compiling separate versions of a subkernel targeting different vector widths. In Section 6.5, we describe how subkernel partitioning reduces the overheads associated with applying the vectorization transformation proposed in Chapter 5.

6.3.1 Impact on Thread Scheduling

Subkernels present an executable structure which enables scheduling to exploit both control-flow uniformity and spatial data locality. Finer scheduling granularity increases spatial and temporal instruction locality by promoting more frequent context switches. In Section 5.2.3, we illustrate several implications of serializing lightweight logical threads for execution by a single hardware

thread which we summarize here. Data parallel kernels are designed to express spatial locality and control-flow uniformity, yet much of this structure is discarded when threads are serialized. Multiple threads performing strided accesses through an array, for example, achieve near-peak throughput during lock-step execution by a GPU. When threads are fused for execution on an ILP processor, thousands or millions of cycles may elapse between one thread accessing an array element before the next thread accesses an adjacent element. Caches lacking capacity or associativity may evict this cache line before the next thread has had the opportunity to access the data. More frequent context switches alleviate this problem by better orchestrating the execution of collections of threads within a kernel, prioritizing the execution of threads within the same region, and context switching when threads exit this region.

Figure 54 provides quantitative measurements of L2 cache performance on the evaluation platform described in Section 6.6. Hardware performance counters show that most subkernel partitioning heuristics tend to reduce L2 miss rates. Benchmarks such as *BinomialOptions*, *MonteCarlo*, *ScalarProduct*, and *SobolQRNG* show subkernels uniformly reduce miss rate. Other benchmarks such as *MersenneTwister*, *Nbody*, and *P-cutcp* illustrate that most partitioning heuristics and warp sizes do reduce miss rate, but there exists a pathological partitioning and warp size that increases it. We attribute the reduction in cache miss rate to more frequent context switches which result in more finely interleaved thread schedules. Coordinated access to the same blocks of memory across threads within the CTA thus result in improved cache utilization.

Because subkernels increase the set of concurrently executable regions within a kernel, this abstraction facilitates alternative mappings of cooperative thread arrays onto hardware threads. Previous work assumes a mapping of one CTA per logical thread which approximates decisions of scheduler logic in today's GPUs. However, CPUs offer more tightly coupled logical processors which may benefit from partitioning the threads of a CTA onto multiple logical threads, executing them concurrently on the same physical processor, and enjoy the benefits of large CPU caches and symmetric multithreading.

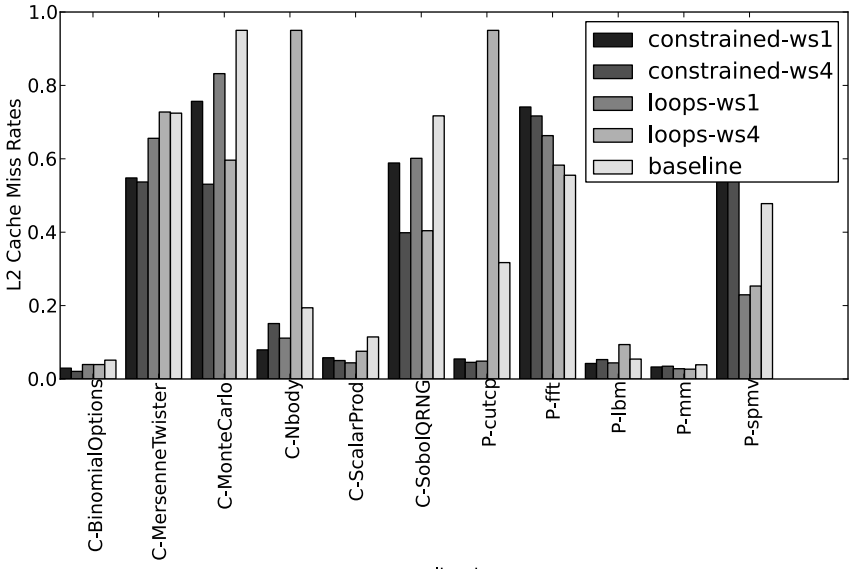


Figure 54: L2 cache miss rates for subkernel partitioning. Baseline performs no context switches except barriers.

6.3.2 Subkernel Definition and Compilation

A **subkernel** is an *induced subgraph* of a compute kernel in which edges spanning subgraphs are treated as context switches. The following definitions hold for subkernels.

- **K** is a compute kernel with control flow graph **G**
- subkernel **S** contains a connected subset of basic blocks in **G**
- (b_i, b_j) is an **internal edge** in **S** if and only if (b_i, b_j) is an edge in **G** for $b_i, b_j \in$ **S**
- (b_i, b_j) is an **external in-edge** of **S** if and only if $b_i \notin S$ and $b_j \in S$
- (b_i, b_j) is an **external out-edge** of **S** if and only if $b_i \in S$ and $b_j \notin S$
- control reaching external edges results in thread context switches

Figure 55 illustrates a simple kernel containing a divergent conditional branch partitioned into two subkernels. PTX within the original kernel blocks in each subkernel is omitted for brevity.

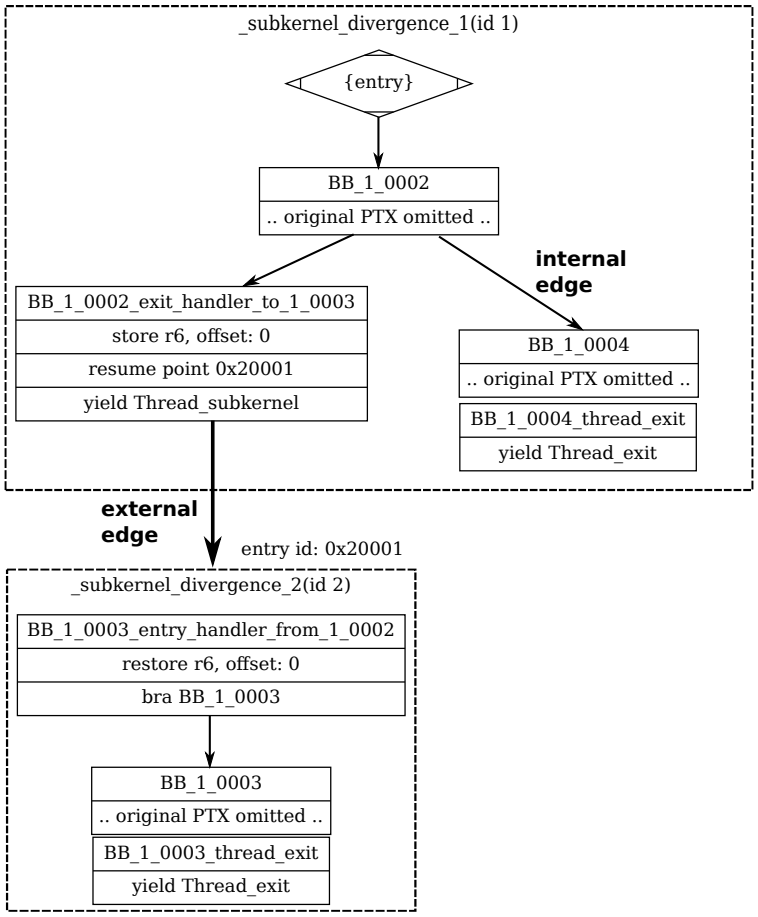


Figure 55: Divergent kernel partitioned into two subkernels with handler blocks inserted along external edges.

Each subkernel is annotated with a globally unique identifier, and entry points within a subkernel are also uniquely identified. Exit handlers which split external edges store live values to thread-local memory, set the thread's *resume point* by encoding the target subkernel and entry IDs, and yield by exiting the subkernel. When control exits the subkernel, the execution manager groups all threads waiting to enter the next subkernel by their resume point and control enters the subkernel targeted by the external edge. Entry handler blocks load values that are live along external edges and needed by the subkernel, then an internal control edge branches to the original basic block for resumed execution.

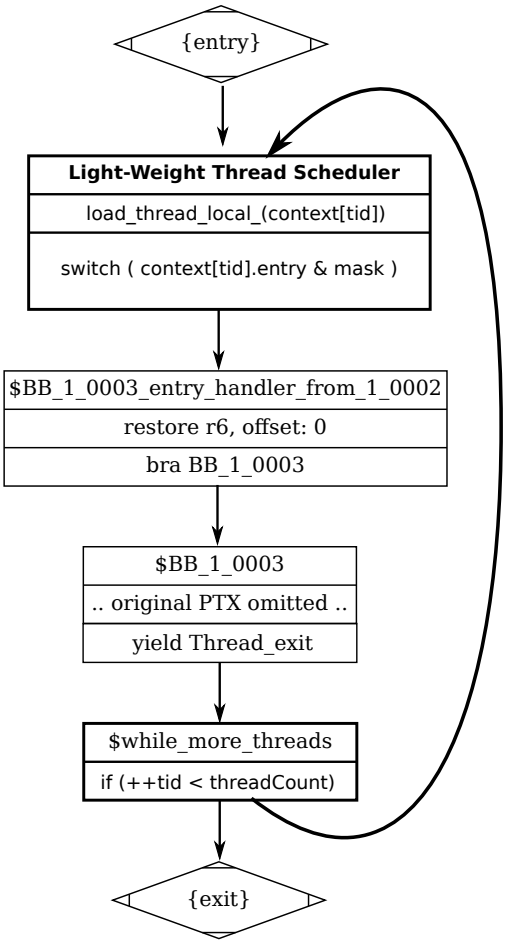


Figure 56: Subkernel transformed by inserting Light-Weight Thread Scheduler and thread loop back edge.

External edges constitute compiler-inserted cooperative yield points within the kernel. Subkernel boundaries may be arbitrarily located within a kernel, but this work proposes and evaluates partitioning heuristics in which subkernel boundaries are placed at specific control-flow instructions. Table 12 summarizes control structures in which subkernel boundaries are placed and indicates special handling logic by the execution manager. All exits require storing live state and updating each thread's entry ID. Certain control structures demand specific handling to accommodate semantics of the execution model. For example, barriers require all threads reach the barrier before any threads may resume execution. Divergence is an event specific to vectorized subkernels in

which conditional control flow across threads is evaluated non-uniformly, and control must exit the subkernel to reform warps.

Table 12: Subkernel thread exit conditions.

Status	Execution manager action
`Entry`	Thread's initial entry into subkernel
`Divergence`	Control-flow instruction non-uniform across threads
`Barrier`	Thread has reached a barrier
`Exit`	Terminate thread
`Subkernel`	Thread has taken an external edge

Figure 56 illustrates the transformations applied within a subkernel. A scheduler block is inserted along the entry edge to load thread-local context such as pointers to shared and local address spaces, thread block dimensions and IDs, and implementation-specific data structures such as a list of texture bindings. The scheduler block then loads the thread's *entry id* which indicates which basic block(s) were the intended successor of an external edge targeting the subkernel. The scheduler is terminated with an indirect branch which jumps to an *entry handler* block which restores live values from thread-local memory. The entry handler then falls through to the target block containing a translated form of the basic block from the original kernel, and execution resumes. Liveness along external edges is determined through data-flow analysis performed on the original kernel prior to subkernel partitioning.

6.3.3 Partitioning Heuristics

Kernels are logically decomposed into a collectively exhaustive set of control-flow graph partitions. Each partition becomes a new subkernel, and control edges between subkernels are replaced with handler basic blocks. This work focuses on the following kernel partitioning heuristics motivated by prior work in characterizing GPU workloads [95].

The **Constrained-with-barriers** heuristic forms subkernels consisting of at least one basic block, constrained to avoid exceeding a certain threshold of PTX instructions. Very large basic

blocks are split, yet most kernels contain sufficiently short basic blocks that this was not necessary for the thresholds we selected.

The **Inner-loops** heuristic avoids splitting inner loops and barriers into separate subkernels. In many kernels, the inner-most loop consists of a subkernel entry at a barrier, computation, subkernel exit at the next barrier, and a back edge through the execution manager which selects a new thread and resumes at the first subkernel entry. This partitioning heuristic performs a depth-first traversal over the tree yielded by control structural analysis [51] and groups leaf nodes into the same subkernel subject to a maximum instruction size. Loop headers mark the creation of a new subkernel ensuring inner loops are grouped together.

The **Maximum-size** heuristic is the baseline heuristic in which all kernels are treated as subkernels and avoids partitioning into smaller regions that are compiled separately. Subkernel exits only take place during the following compulsory events: barriers, function calls, divergent control flow in vectorized code.

6.4 Implementation

Subkernels enable an execution manager to select a set of ready threads and enter a subkernel. A compilation transformation inserts a Light-weight Thread Scheduler (LWTS) into the subkernel which loops over thread contexts and executes the subkernel for this set of threads to completion. Threads which take an external edge suspend their state, and control returns to the light-weight scheduler which selects the next thread or exits the subkernel entirely. Upon exiting the subkernel, control returns to an execution manager which forms a new set of threads. Two-level scheduling with subkernels permits fast context switching over collections of threads and sophisticated warp formation and thread selection at coarse boundaries such as subkernel external edges. For example, if a collection of threads reaches a barrier and control exits the subkernel, program correctness asserts that all other threads must execute the same subkernel and so the next collection of threads can be formed from all remaining threads in the CTA. As another example, if divergence analysis

indicates all control decisions within a subkernel are uniform, then only one specialized version of the subkernel need be JIT compiled.

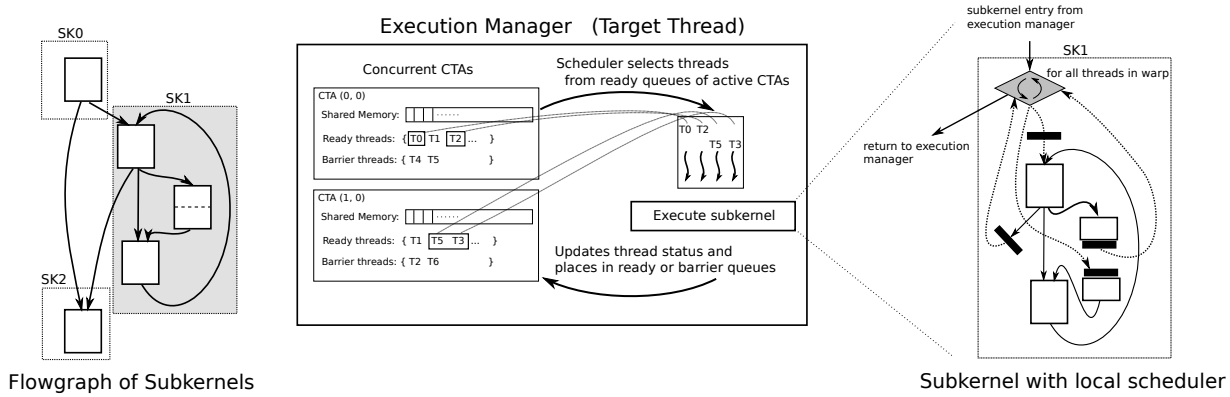


Figure 57: Subkernel partitioning, dynamic compilation, and execution. A kernel partitioned into subkernels is executed via an execution manager running in a host thread.

Figure 57 illustrates the structure of a kernel decomposed into subkernels and executed by an execution manager running in a target thread. The execution manager receives a set of CTAs and selects a subset of these to execute concurrently. From this subset, it chooses a set of *ready* thread contexts waiting to execute the same subkernel and queries the dynamic translation cache. A set of worker threads running on local CPU cores invoke an execution manager, which is responsible for dynamically mapping the abstractions in the PTX execution model onto the computational resources available as follows. The kernel's grid of CTAs is distributed across them dynamically using work-stealing. The execution manager orchestrates the execution of all PTX threads from this set of CTAs while respecting CTA-wide barrier semantics. The execution manager is free to interleave the executions of threads from multiple CTAs arbitrarily because the PTX execution model does not define inter-CTA synchronization and data sharing.

Code is translated and cached at the granularity of subkernels. During execution subkernels are fetched and translated as needed. When PTX threads are scheduled on a subkernel that is not present in the local cache, the global JIT compilation facilities are queried which may have a cached translation of the desired region for the correct warp size. If not, the desired subkernel,

warp size, and other specialization parameters are queued by the dynamic translation cache. When no other threads can be found that are ready to execute, the thread is stalled during the execution of the CTA and devoted to translation, optimization, and code generation for the queued subkernels. This approach (1) allows most transitions to new subkernels to avoid inter-thread communication, (2) avoids evicting large blocks of live application state from the cache hierarchy, (3) delays compilation until the demanded region is known to be needed, and (4) enables translated code to be shared among multiple worker threads. The size of subkernels, controlled by the active partitioning heuristic, can be used to balance the overheads of inter-thread communication against the cost of dynamically compiling dead code.

Within the subkernel, an indirect branch based on the warp's entry point identifier transfers control from the subkernel entry to the actual target block the warp is waiting to enter. All edges (dotted) from the subkernel scheduler to instruction blocks pass through a restore point which loads live registers from thread-local memory. Control flow within the subkernel blocks are unaffected, but edges that exit the subkernel are replaced with store points and returns to the local scheduler. The Light-Weight Thread Scheduler may execute a different thread in the warp or return to the execution manager. If threads have exited a subkernel at a barrier, their context objects are placed in their owning CTA's barrier queue. If threads reach a kernel exit, they are removed from the ready queue entirely. Otherwise, they are returned to the ready queue, and a new warp is formed. When a CTA has no threads in its ready queue, the threads in the barrier queue are moved to the ready queue. If the ready queue is still empty, the CTA is retired and memory is reclaimed. In practice, this data structure is implemented as a circular buffer, and queue operations are performed in constant time.

6.5 Subkernel Optimizations

The execution manager running in each worker host thread chooses the set of PTX threads that execute a subkernel - *dynamic warp formation*. The goal of warp formation is to improve execution

performance by selecting a small subset of threads for lightweight fine-grain scheduling within the subkernel. This scheduling takes advantage of (1) instruction locality, as threads must execute the same subkernel, (2) memory locality, as the programming model recommends adjacent threads access the same regions of memory, and (3) control-flow uniformity, as programmers can achieve high utilization by reducing thread divergence. In this work, we implement and apply several aggressive program transformations that exploit these characteristics of PTX kernels. Specifically, we leverage control-flow and memory locality to vectorize PTX kernels and eliminate redundant expressions via affine analysis.

Affine Analysis. Subkernels present regions for targeting specific dynamic optimizations such as the above vectorization which transforms a region of scalar instructions into equivalent vector instructions. Execution within vectorized subkernels is logically equivalent to the execution of multiple scalar threads within a single instruction sequence. *Affine analysis* is an additional optimization which statically determines which expressions, if any, are linear combination of some uniform value and thread ID added to a thread-invariant value. For example, each thread may compute an address from a base value passed as a kernel parameter added to the product of thread ID and size of the data type to be written. Sampaio et al. [134] describe a static compiler analysis using gated static-single assignment form for detecting thread invariance and affine expressions in PTX kernels.

Vectorization. In the context of data-parallel compute kernels, *vectorization* as described in [90, 98] refers to a program transformation in which a scalar function or kernel is replicated such that a single execution of the vectorized program is logically equivalent to the execution of the scalar program for multiple threads or data elements. Traditional approaches to vectorization transform the iteration space of loop nests such that multiple iterations are executed by the lanes of SIMD functional units. In this work, vectorization refers to transforming scalar kernels with respect to the PTX execution model, replicating and interleaving instructions such that executing a vectorized subkernel is equivalent to executing the same scalar subkernel for multiple threads.

Affine analysis enables identifying memory accesses in which threads of a warp access consecutive locations. We use this to conservatively select load and store instructions which may be replaced with a single vector load or store to a base address satisfying the entire warp. Redundant address computation instructions are elided. This work applies the algorithms developed in [98] to subkernels, as they offer variable granularities of specialization. This implementation makes several passes over the subkernel's internal representation and presents a considerable overhead in dynamic compilation.

6.6 Experimental Evaluation

This section presents the results from an evaluation subkernel formation and execution. This concept was implemented as a device back-end for GPU Ocelot-2.0.1894 [48] compiled with LLVM 3.2svn [105]. Evaluations were conducted on a desktop workstation running Ubuntu 12.04 x86-64. The evaluation system contains an Intel Sandybridge (i7-2600) CPU supporting SSE 4.2 and AVX. Workloads chosen from the CUDA Software Development Kit, the Parboil Benchmark Suite [81], and the Rodinia Benchmark suite [26]. Results are gathered by running each application ten times on the test platform in isolation and computing the arithmetic average of each measurement across all kernel executions within each application. Hardware performance counters instrumented with PAPI [20] were used to obtain cache and IPC results.

Execution Coverage This experiment identifies what fraction of PTX subkernels are actually required for executions using existing data sets for several real-world applications. The results are illustrated in Figure 58, presented as fraction of subkernels that are actually dynamically compiled and executed relative to the total number of partitions. Results are plotted for two heuristics - constrained and loops as defined in Section 6.3.3 - and two maximum warp sizes: 1 and 4. Kernels with more dead code show lower execution coverage. This is particularly apparent when the maximum warp size is set to 4 threads, as kernels with highly convergent control-flow behavior never require compiling the scalar specialization of subkernels.

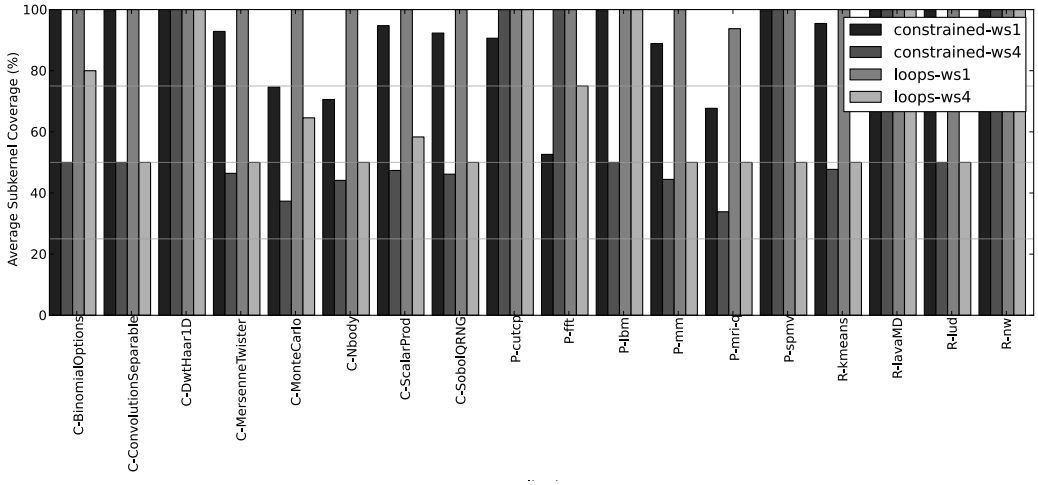


Figure 58: Average subkernel execution coverage for several warp sizes and heuristics.

Additionally, results indicate that many applications demonstrate reductions in compiled code size through subkernel partitioning even for a warp size of 1. Parboil's *P-fft* application demonstrates that approximately half of the kernel is dead for datasets provided, though most applications show more code liveness. Rodinia's *R-nw* compiles all specializations of all subkernels for both heuristics indicating all code paths have been executed. `constrained-ws1` yields an average of 90% subkernel coverage, and `constrained-ws4` yields an average of 64% indicating a substantial reduction in translation cache size.

Startup Just in time compilation imposes runtime costs, when a kernel that has not yet been translated is executed and the runtime must compile it. Subkernel partitioning reduces the granularity of compiled kernels and enables overlapping compilation with execution. Additionally, subkernels avoid compiling all parts of heavily inlined kernels with significant regions of dead code in control paths that are never taken. This experiment measures time to execute the first invocation of each subkernel within an application and compares the results with static compilation.

Figure 59 illustrates the latency in first executing a translated kernel for several partitioning heuristics and warp sizes. This is normalized against full kernel JIT compilation. Most applications show substantial reductions in initial startup time, and this reduction becomes more significant for

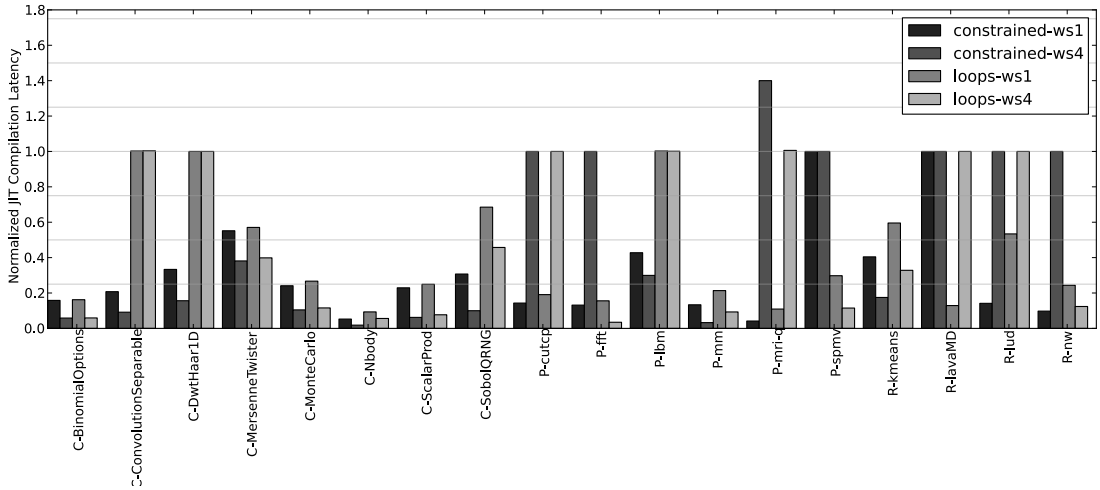


Figure 59: Kernel startup latency.

larger warp sizes. This result is consistent with aggressive optimizations with runtime proportional to program size. A curious outlier is Parboil's *P-lbm* benchmark which shows no improvement with region-based compilation, and vectorizing for warp size of 4 threads makes startup overheads worse than the baseline. This is due to the single CUDA kernel consisting of a single acyclic region, and the `inner-loops` heuristic creates a subkernel that is identical to the original kernel.

Kernel Runtime This benchmark measures the runtimes of kernels using several subkernel partitioning heuristics and warp sizes for fixed-size data sets. This measurement includes dynamic compilation overheads, as subkernels are compiled lazily as demanded. These results are presented in Figure 60. The baseline performance assumes no partitioning but compares against the same warp size. The best performance results show `constrained-ws4` achieves a normalized execution time of 0.93x. The results indicate that many applications benefit substantially for smaller compilation granularities such as CUDA SDK's *MersenneTwister* application and Rodinia's *R-lud* benchmark for warp size of 4.

Steady-State Kernel Runtime This metric measures strictly the performance of statically compiled kernels using each partitioning heuristic for various warp sizes. The results are plotted in Figure 61 and capture strictly the effects of subkernel partitioning and thread scheduling

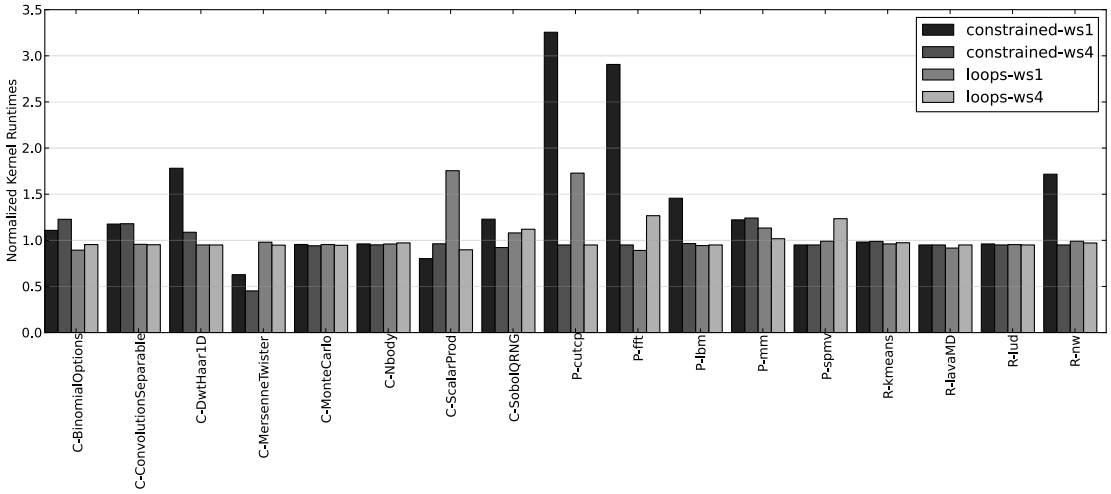


Figure 60: Normalized kernel runtimes with subkernel partitioning and lazy compilation.

on execution time. All subkernels are compiled before any timing measurements take place. The baseline performance assumes no partitioning but compares against the same warp size. constrained-ws4 achieves an average of 87% normalized runtime compared to `maximum-ws4`, whereas `loops-ws4` shows an average of 91% normalized runtime. Several applications perform considerably slower with some forms of partitioning such as *P-cutcp* and *P-fft*, and these would benefit with no partitioning at all.

Distribution of Execution Time This metric determines the distribution of execution time during the execution of kernels by the execution manager for three partitioning heuristics: constrained, loops, maximum. The results are illustrated in Figure 62. Each application corresponds to three stacked bars totaling 100%. Execution time is divided among **(1.)** initialization - time spent configuring the kernel for launching - **(2.)** compilation - time spent performing dynamic compilation - **(3.)** scheduling - time spent managing thread contexts, accommodating barriers, and selecting subkernels - and **(4.)** execution - time spent within the subkernel. Each application illustrates the distribution of runtime for each partitioning heuristic. The results show subkernel partitioning affects how runtime is distributed among several phases of executing CTAs. However, the set of benchmarks react differently and do not show the same trends between partitioning heuristics.

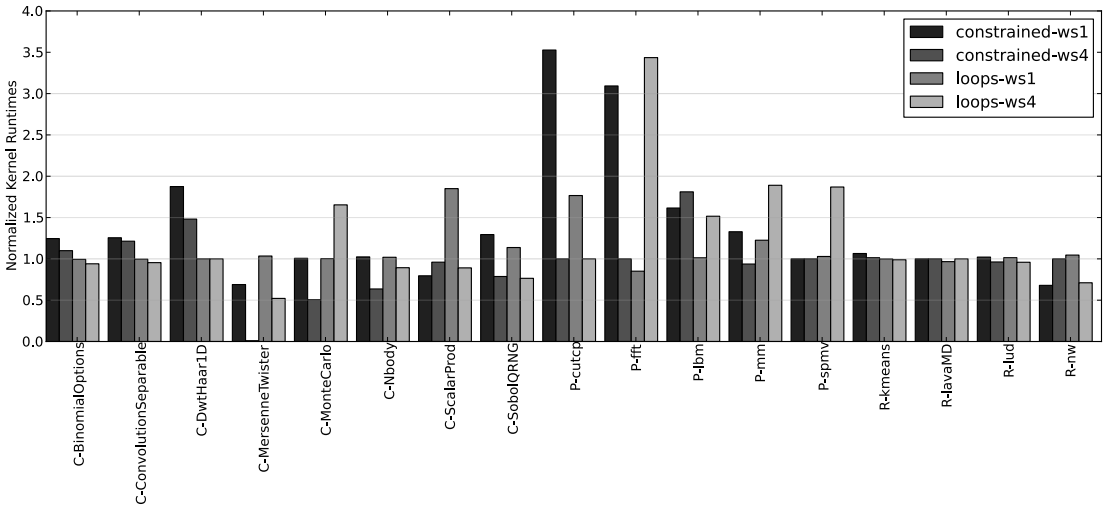


Figure 61: Normalized steady-state kernel execution with subkernel partitioning and eager compilation.

Subkernels reduce the size of compiled code regions, avoiding compiling large regions of dead code which may be incurred in heavily inlined kernels, and facilitate specialization while avoiding significant code expansion. However, this technique inserts additional thread context switch locations along subkernel region boundaries and increase the frequency of context switches. As prior work has demonstrated [49, 68], there is a performance overhead in performing context switches due to additional instructions and memory transfer necessary to store and restore thread state in addition to overhead in the execution manager. Thus, intuition suggests performance should be negatively impacted. Moreover, most kernel-oriented programming models place importance on achieving high throughputs at the expense of startup costs and amortize JIT compilation costs over long-running executions. This motivates the application of expensive and aggressive optimizations that improve performance by even moderate amounts, as the speedup over long computations makes them worthwhile.

Our results demonstrate performance gains even when compilation is performed *a priori*. We attribute this to better utilization of the memory hierarchy, with more frequent light-weight context switches better able to capture temporal and spatial locality among threads. We also note that many

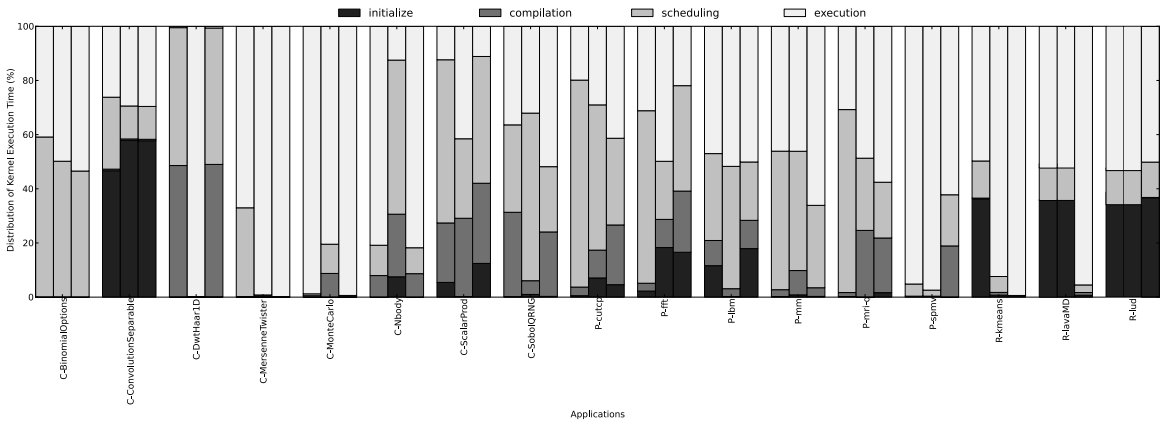


Figure 62: Distribution of kernel execution time for partitioning heuristics: constrained, loops, maximum.

kernels include large basic blocks with barrier synchronizations within their inner loops as threads cooperatively perform a set of computations, synchronize, and exchange data thus incurring nearly as many context switches as implied by subkernel partitioning. We admit neither of the partitioning heuristics produces maximum speedup for all applications, and several benchmarks perform best with no partitioning. Statically determining the best partitioning heuristic remains future work. This research does not apply aggressive specialization beyond vectorization. We cite aggressive predication [90] and parameter-based specialization [99] as immediate applications of this work.

6.7 Related Work

Hank et. al [73] describe initial applications of region-based compilation toward exploiting instruction-level parallelism on VLIW processors for single-threaded applications. Region-based compilation has been explored in several traditional execution environments to address overheads and costs in performing compilation at runtime [146]. In particular, region-based compilation enables aggressive inlining and interprocedural analysis. The Azure dynamic binary translator [157] infers control-flow graph representations from binary analysis and proposes a partitioning process

into single-entry, multiple-exit "superblocks" to enable vectorization and loop-level parallelization. Like this work, Azure relies on an execution manager to monitor control edges and, when control flow exits analyzed regions, the execution manager infers the next region of the program to begin executing. Unlike previous instantiations of dynamic compilers and managed runtimes, this work leverages explicit parallelism and application characteristics to achieve high performance on many-core processors. Moreover, subkernels present advantages both as dynamic compilation abstractions as well as thread scheduling abstractions for fine-grain cooperative multithreading.

Several frameworks have been proposed for statically compiling and executing GPU computing applications on multicore CPUs. MCUDA [144] defines a source-to-source compilation technique that transforms CUDA to C++ through inserting thread loops and expanding scalar variables to arrays. Stratton et al. [143] describe an alternative method for implementing fine-grain threading through compiler-inserted goto statements in the source representation of the kernel. Lee et al. [106] describe an implementation that compiles OpenCL for execution on Cell Broadband Engine processors and propose work-item coalescing and a novel data flow analysis technique for selecting values to store during context switches. In Chapter 5 of this thesis, we present insights related to execution model transformation in compiling PTX to x86 ISA, particularly that infrequent barriers affect memory access patterns. AMD's Twin Peaks OpenCL to multicore CPU compiler [68] includes a highly optimized *setjmp(), longjump()* implementation that improves performance for barrier-intensive OpenCL kernels when executing on x86 processors. They also investigate the impact of user-level threads on the memory hierarchy and apply several optimizations such as skewing thread local memory offsets and selecting large pages to improve TLB hit rates.

Optimization techniques for bulk-synchronous parallelism have been proposed. Guo et al. [69] describe analysis and program transformations to identify instances in which inter-thread data-flow semantics were intended by the programmer and treated correctly without incurring the overhead of a synchronization. This technique assumes a static mapping of threads to thread loop iterations and does not correctly handle the case in which implicitly synchronized code appears in divergent

regions. Accommodating SIMD processors is a natural application for data-parallel programming models, yet achieving portability with flexible programming models such as OpenCL and CUDA has remained a challenging problem without substantial hardware modifications. Efforts to replace control-flow with data-flow have achieved some success [90] in compiling OpenCL to x86 with SSE. Specialized languages [110] leverage constraints in the programming model to perform several program and data layout transformations such as vectorization and transformation to structure-of-arrays.

6.8 Conclusion

Subkernel partitioning improves compilation granularity enabling a reduction in startup time due to dynamic compilation without incurring a relative increase in overall kernel execution time. When coupled to optimizations such as affine analysis and specializing for vectorization, overall performance improves significantly even for small datasets. Thus, we present a technique to improve the state of the art for dynamic compilation of kernel-oriented programming models without sacrificing either performance or portability. Moreover, for thread fusion techniques relying on compiler-inserted yield points for cooperative multi-threading, the increase in scheduling frequency actually improves the performance of kernels that exhibit lock-step behavior and spatial memory access locality. Optimizing compilers that aggressively apply specialization benefit in particular from avoiding the need to compile multiple specializations of entire kernels when smaller regions suffice.

CHAPTER VII

ENGINEERING A HETEROGENEOUS COMPILER

The development of GPU Ocelot placed strong design goals in separating device-specific components from device-agnostic intermediate representations and API front ends. As a compiler, GPU Ocelot provides several points in the compilation flow to insert analysis and transformation procedures that reflect this separation. As an execution manager, GPU Ocelot provides several mechanisms to observe and modify the behavior of applications as they are running to gain insight into their performance characteristics and tune their executions for improved efficiency. To the best of our knowledge, this work is the only dynamic compilation framework supporting both high-end commodity GPU architectures and multicore CPU architectures. This chapter describes the design decisions made during the development of GPU Ocelot, presents its internal structure, and evaluates the performance of each backend for a common set of workloads. This chapter is expected to be of use to potential users of GPU Ocelot or designers of future dynamic compilation and runtime infrastructures with a related set of constraints and design goals.

7.1 Intermediate Representation

GPU Ocelot's PTX intermediate representation (PTX IR) provides a low-level representation of data parallel kernels on which compiler analysis and program transformations may be implemented. Ocelot provides capabilities such as rewriting PTX modules dynamically, translating other languages to PTX, translating PTX to other languages, and JIT compiling kernels for execution on each device backend. GPU Ocelot inputs PTX via a textual representation embedded in CUDA application binaries. Applications register PTX modules as *cubin* objects, data structures containing both opaque references to a collection of compiled binaries as well as a string format containing

PTX. Ocelot parses this PTX into its own IR and registers the resulting module with several devices which have the option of eagerly translating and JIT compiling the PTX module immediately or lazily storing a reference to the module. The PTX IR implements emitters to text for obtaining a validated string representation of PTX operands, PTX instructions, and whole modules which may contain definitions of multiple kernels, textures, and global variables.

Control-flow graph. The control-flow graph is the fundamental data structure describing compute kernel implementations. GPU Ocelot implements an ISA-agnostic directed graph data-structure storing a dense list of instructions per node, called a *basic block*, as well as annotated edges indicating whether the control edge is the result of a branch, a fall-through edge resulting from adjacency, or a dummy edge used to ensure correct data-flow semantics during subsequent analysis and transformation passes. GPU Ocelot's control-flow graph container offers numerous iterators such as breadth-first traversals over blocks and instruction iterators over the instructions within blocks. Mutators facilitate adding and removing basic blocks, creating new edges, splitting edges to insert new blocks, and splitting blocks to create new edges.

Compute kernels. Kernels store declarations about input and output parameters, name and module visibility, whether they are device-only, local variable declarations, and finally the control-flow graph of the function they define. Kernels are defined for each device and enable device-specific transformations which may be necessary for each device backend. For example, the PTX emulator backend requires a dense packing of PTX instructions such that fall-through edges between basic blocks imply consecutive layout in memory. For the NVIDIA backend, this requires an additional textual representation of the PTX module which may be loaded via the CUDA Driver API [126] and JIT compiled. For the multicore backend utilizing the region-based compilation technique described in Chapter 6, the compute kernel maintains a partitioned graph of subkernels. Additional details are described in Section 7.3.

Variable Definitions. PTX specifies that variable declarations may occur in either module or kernel scope. These variables may be instantiated in one of `.global`, `.local`, or `.shared`

state spaces, with additional semantics implied by the particular combination of properties. For example, a module-scoped `.shared` variable declared with `extern` is effectively treated as a pointer to the beginning of shared memory and is needed to compute pointers to additional shared variables.

PTX Instructions. PTX instructions maintain a set of opcodes, modifiers, and operands representing all of the instructions in the PTX specification. The PTX emulator emits a stream of PTX instruction instances via its trace generator interface which are consumed by attached analysis tools for various purposes such as (1.) workload characterization, (2.) driving cycle-accurate simulators, and (3.) validating program behavior. The PTXInstruction class defines a compact representation of instructions which may be conveniently accessed by other procedures, emitted as parseable strings, and validated.

The PTX IR used for this project is intended to serve as the front-end interface to GPU Ocelot, the source for translation to other devices, the executable representation for the PTX emulator, and the internal representation for compiler analysis and transformation.

7.1.1 Critique

Utilizing the same data structures for the target of parsing, analysis, transformation, translation, and direct execution is a challenging proposition. No apparent design satisfies all purposes ideally. The implementation adopted for GPU Ocelot has a flat class hierarchy and compact representation, particularly for representing PTX instructions. Enumerated types, associative arrays, and accessor methods make Ocelot's PTX IR a succinct representation for interacting with PTX. GPU Ocelot's PTX parser faithfully accepts nearly all statements and declarations within PTX modules including `.loc` directives indicating where in the original CUDA source file a particular block of PTX instructions correspond to. The only information lost between parse and emitting PTX are comments, extraneous directives, and variable names. The latter are replaced during register allocation within the CUDA device driver and do not correspond to physical resources. GPU Ocelot's parse

and emit phases are idempotent.

As a representation for compiler analysis, data-flow analysis is expressed as an overlay data structure on the `ControlFlowGraph` and includes mutators for adding and removing instructions while updating definition-use lists and preserving static-single assignment form if need be. Adding instructions directly to BasicBlock instances within the control-flow graph creates an inconsistent view of the kernel and implicitly invalidates existing DFG instances. Unfortunately, the code base does not fully encapsulate program objects to prevent inconsistencies among overlay data structures.

Secondly, the PTXInstruction class is the only class for interacting with PTX instructions and includes attributes and modifiers for all possible instructions. Thus, GPU Ocelot's IR makes it possible to set attributes that have no bearing on the given instruction opcode. Moreover, it is possible to assign operands that are invalid given the opcode, such as attempting to the store the results of an `add.u32` instruction to an address or program label. Program validation is built into the PTXInstruction class to raise exceptions when invalid PTX instructions are created, but module-level checking is not currently implemented. Robustly checking types and eliminating implicit casts are also not supported.

These shortcomings underscore the challenges in using a single IR as both an executable representation and as an intermediate representation for compiler analysis. Executable representations require compact structures with immediate values that avoid traversing linked structures for every attribute or value. Flat class hierarchies and fully visible opcodes and modifiers enable simulator implementations to quickly decode instructions. This is especially beneficial to GPU Ocelot's PTX emulator which frequently executes the same decoded instruction for multiple logical threads. Alternatively, deep class hierarchies with runtime type decoding enable abstract reasoning about program structures, eliminate the possibility of constructing overdetermined or invalid programs, and facilitate implementations of compiler analysis and transformations.

7.2 API Frontends

NVIDIA's CUDA Programming Language has emerged as popular model for developing applications utilizing accelerator devices while expressing computational parallelism in a manner that may be conveniently mapped onto coarse-grain and fine-grain cores. GPU Ocelot extends this concept to support multiple Application Programming Interfaces (APIs), and multiple device backends each targeted by a common intermediate representation expressed with a common execution model. This vision applies dynamic compilation and execution model transformations to achieve efficient execution on each device type.

7.3 Device Interface

The design goals of GPU Ocelot as a device-agnostic compilation framework necessitate modularity and well-designed interfaces between components. These avoid unnecessary coupling between layers and facilitate the addition of new devices, new API front-end layers, and optional optimization and transformation passes. This relationship between APIs, the device interface, and device backends is illustrated in Figure 63. This section describes the device interface and each of the device backends implementing it.

7.3.1 PTX Emulator

Detailed workload characterization, application profiling, and trace generation for precise timing models require access to machine state, whether physical or simulated. This motivated the development of GPU Ocelot's PTX emulator device backend.

Functional Simulation. The PTX emulator is designed to correctly satisfy the PTX execution model without attempting to replicate actual GPU microarchitectural details. To faithfully model a multiprocessor pipeline would require, among other things, modeling instruction latencies, L1 cache, on-chip interconnect, the hardware thread scheduler, and memory controllers. Several excellent GPU and heterogeneous cycle-level simulators are available such as MACSIM [2],

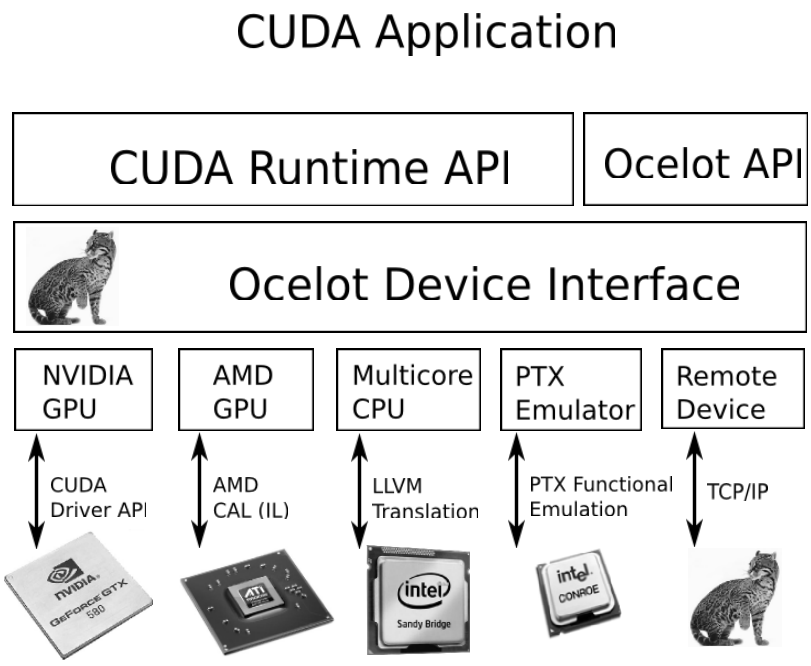


Figure 63: GPU Ocelot device interface.

GPGPU-Sim [9], and Rigel [91]. Rather, the PTX emulator is a functional simulator producing correct instruction traces from the perspective of each thread assuming deterministic application behavior. This ignores circumstances in which timing affects program outcome such as contention for locks and data races. This realization also ignores possible concurrency among CTAs and data flow between them. Both of these decisions are supported by elements of PTX execution model specification, though some parts of PTX expose hardware realizations from NVIDIA.

To the best of our knowledge, GPU Ocelot's PTX emulator device backend satisfies the strict requirements of the PTX execution model but contradict numerous properties of GPU hardware realizations. For example, many high-performance implementations of prefix scan and other reductions on GPUs assume warp-synchronous execution of multiple threads within the same warp and omit synchronization primitives when these threads exchange data through shared memory. On GPU hardware, this behavior has remained consistent across three successive GPU microarchitectural revisions from NVIDIA. Nevertheless, the programming model does not provide a way

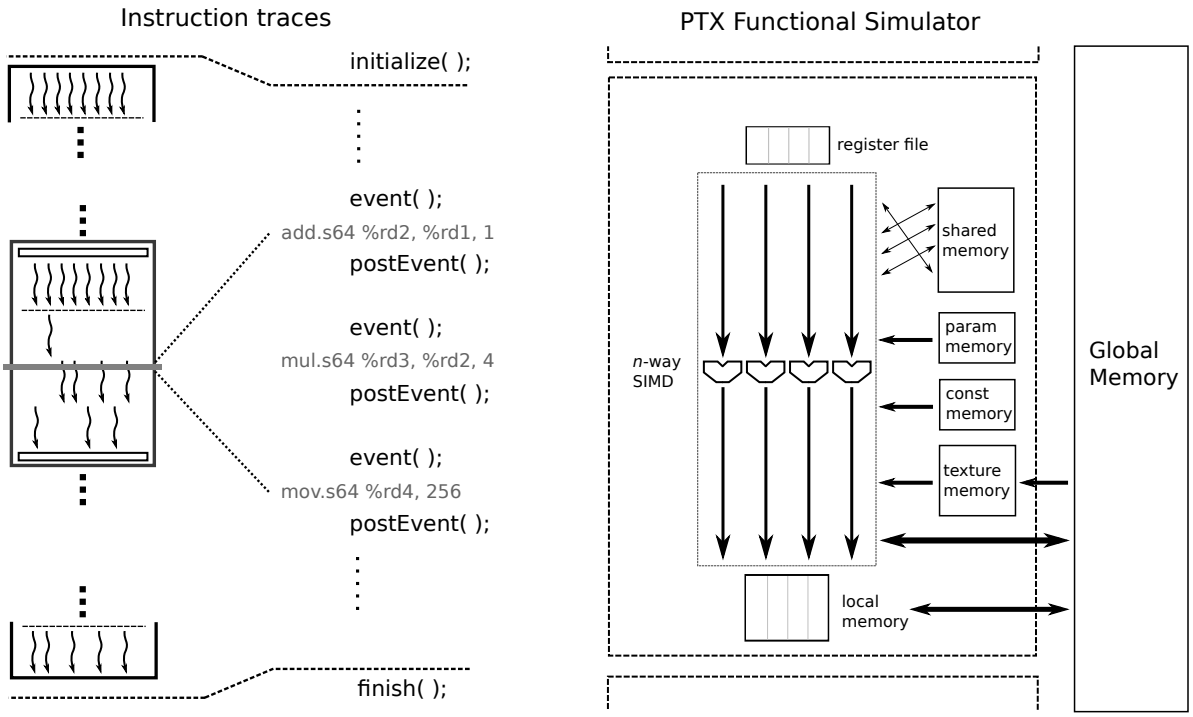


Figure 64: PTX Emulator trace generation facilities with abstract machine model.

to assert to the compiler which threads are guaranteed to reach a given block of code in lock-step nor is there a clear way to enforce concurrent execution among CTAs. Some work attempts to use static data-flow analysis to identify implicit data dependencies among different threads assuming this lock-step execution model [69] but even this technique cannot accommodate all possible programs, particularly if the implicitly synchronized regions are reachable by diverged warps with data-dependent control properties. In implementing a 3D rasterization pipeline in CUDA [104], the authors note contended access to shared memory has the property that the thread executing in the highest-numbered lane always writes its value last and may be relied upon for order-dependent triangle rasterization.

The emulator cannot properly model time, but it can model the architectural state of a model GPU processor core during the execution of kernels. The complete architectural address space of a multitprocessor is modeled which includes the following: the register file, shared memory, local

memory, global memory, and texture bindings. To simplify the implementation of the emulator and to avoid encumbering the authors of analysis tools with many of the challenges associated with multi-threaded programming, execution within the emulator is entirely serialized. And yet, several notions of parallelism are preserved.

SIMD parallelism is preserved by executing one instruction for all threads at this program counter before moving to the next instruction. This enables the emulator to accurately model thread reconvergence techniques. The emulator's implementation of reconvergence is abstracted by the `ReconvergenceMechanism` class declared in Listing 7.1. Researchers may implement this abstract base class in accordance with their experimental reconvergence mechanism and thread scheduling policy. Natively, GPU Ocelot implements immediate post-dominator reconvergence [59] and reconvergence only at thread barriers. Thread frontiers [47] describes a priority-based scheduling technique in which all threads have their own program counter, and the set of threads with the highest-priority program counter execute while the rest are stalled. Several implementations are provided.

```
class ReconvergenceMechanism {
public:
  ReconvergenceMechanism(CooperativeThreadArray *cta);
  virtual ~ReconvergenceMechanism();

  /*! \brief initializes the reconvergence mechanism */
  virtual void initialize() = 0;

  /*! \brief updates the predicate mask of the active context before instructions execute */
  virtual void evalPredicate(CTAContext &context) = 0;

  /*! \brief implements branch instruction and updates CTA state
    \return true on divergent branch */
  virtual bool eval_Bra(CTAContext &context,
    const ir::PTXInstruction &instr,
    const boost::dynamic_bitset<> & branch,
    const boost::dynamic_bitset<> & fallthrough) = 0;

  /*! \brief implements a barrier instruction */
  virtual void eval_Bar(CTAContext &context, const ir::PTXInstruction &instr) = 0;

  /*! \brief implements reconverge instruction */
  virtual void eval_Reconverge(CTAContext &context, const ir::PTXInstruction &instr) = 0;

  /*! \brief implements exit instruction */
  virtual void eval_Exit(CTAContext &context, const ir::PTXInstruction &instr) = 0;

  /*! \brief implements vote instruction */
  virtual void eval_Vote(CTAContext &context,
```

```
    const ir::PTXInstruction &instr);

  /*! \brief updates the active context to the next instruction */
  virtual bool nextInstruction(CTAContext &context, const ir::PTXInstruction &instr,
    const ir::PTXInstruction::Opcode &) = 0;

  /*! \brief gets the active context */
  virtual CTAContext& getContext() = 0;

  /*! \brief gets the stack size */
  virtual size_t stackSize() const = 0;

  /*! \brief push a context */
  virtual void push(CTAContext&) = 0;

  /*! \brief pop a context */
  virtual void pop() = 0;
};
```

Listing 7.1: ReconvergenceMechanism interface.

Trace Generation. Obtaining detailed instruction traces as well as providing flexible and straightforward access to the internal device state of the emulator motivated the design and specification of the *trace generator interface*. This enables applications to register a set of call-backs to GPU Ocelot which are invoked during specific points in the execution of kernels. Listing 7.2 declares the abstract base class for trace generators and presents four virtual functions called during the execution of kernels.

`initialize()` is passed a reference to `ExecutableKernel`, the base class for kernel objects which stores its PTX representation, launch configuration, parameter values, and a reference to the active device. This enables trace generators to initiate compiler analysis, inspect startup state and working set prior to kernel execution, and optionally start any timers or initialize trace serialization stores such as opening files or connecting to a database. This method is called by each Ocelot device backend and provides a convenient way of transparently analyzing and monitoring kernel execution.

`event()` and `postEvent()` are called by the PTX emulator backend only during the execution of PTX instructions. These receive a constant reference to a `TraceEvent` instance which completely defines the execution of a single PTX instruction. `event()` is called after memory references have been computed but before any results are committed enabling guarded access to

certain resources, such as checking memory addresses target valid allocations, and to inspect device state. `postEvent()` is called after results are committed enabling trace analysis tools to observe written results such as values written to address spaces or to the register file.

Because these methods require precise and detailed access to the internal state of the abstract machine model being emulated, they are only called by the PTX emulator backend. If a kernel is executed on a different device type, they are never called. It may be possible to modify the multicore CPU backend to preserve references in the translated PTX instruction stream and issue calls from JIT-compiled code executing natively. The dynamic instruction and memory traces would reflect transformations to the execution model such as reordered memory access and smaller warp sizes, and there would be an impact on performance. Nevertheless, the resulting workload characteristics would be accurate depictions of the applications and runtimes are likely to be faster than if executed on the emulator. This remains future work, however.

`finish()` is called immediately after kernel execution halts enabling active trace generators to terminate timers and output their results to data stores of their choosing. This method is called by each Ocelot device backend and provides a convenient way of transparently analyzing and monitoring kernel execution.

```
class TraceGenerator {
public:
  TraceGenerator();
  virtual ~TraceGenerator();

  /*! \brief called when a traced kernel is launched to retrieve some
      parameters from the kernel */
  virtual void initialize(const executive::ExecutableKernel& kernel);

  /*! \brief Called whenever an event takes place. */
  virtual void event(const TraceEvent & event);

  /*! \brief called when an event is committed */
  virtual void postEvent(const TraceEvent & event);

  /*! \brief Called when a kernel is finished. There will be no more
      events for this kernel. */
  virtual void finish();
};
```

Listing 7.2: TraceGenerator interface.

Virtual methods `event()` and `postEvent()` defined in `class TraceGenerator` receive a `TraceEvent` instance as their single argument. This object captures the execution of a PTX instruction. Its declaration appears in Listing 7.3 with members describing kernel launch dimensions, program counter, a reference to the PTXInstruction instance being executed, and a vector of memory addresses that may be referenced by instructions accessing one of several address spaces.

```
class TraceEvent {
public:
  typedef std::vector< ir::PTXU64 > U64Vector;
  typedef boost::dynamic_bitset<> BitMask;

public:
  TraceEvent();

  TraceEvent(
    ir::Dim3 blockId,
    ir::PTXU64 PC,
    const ir::PTXInstruction* instruction,
    const boost::dynamic_bitset<> & active,
    const U64Vector & memory_addresses,
    ir::PTXU32 memory_size,
    ir::PTXU32 ctxStackSize = 1);

  /*! resets instruction-specific events to their 'off' state */
  void reset();

public:

  /*! ID of the block that generated the event */
  ir::Dim3 blockId;

  /*! PC index into EmulatedKernel's packed instruction sequence */
  ir::PTXU64 PC;

  /*! instruction const pointer to instruction pointed to by PC */
  const ir::PTXInstruction* instruction;

  /*! bit mask of active threads that executed this instruction */
  BitMask active;
  /*!  Taken thread mask in case of a branch */
  BitMask taken;
  /*! Fall through thread mask in case of a branch */
  BitMask fallthrough;

  /*! vector of memory addresses possibly generated for this instruction */
  U64Vector memory_addresses;

  /*! vector of sizes of memory operations possibly issued by this  instruction */
  ir::PTXU32 memory_size;

  /*! dimensions of the kernel grid that generated the event */
  ir::Dim3 gridDim;

  /*! dimensions of the kernel block that generated the event */
  ir::Dim3 blockDim;
```

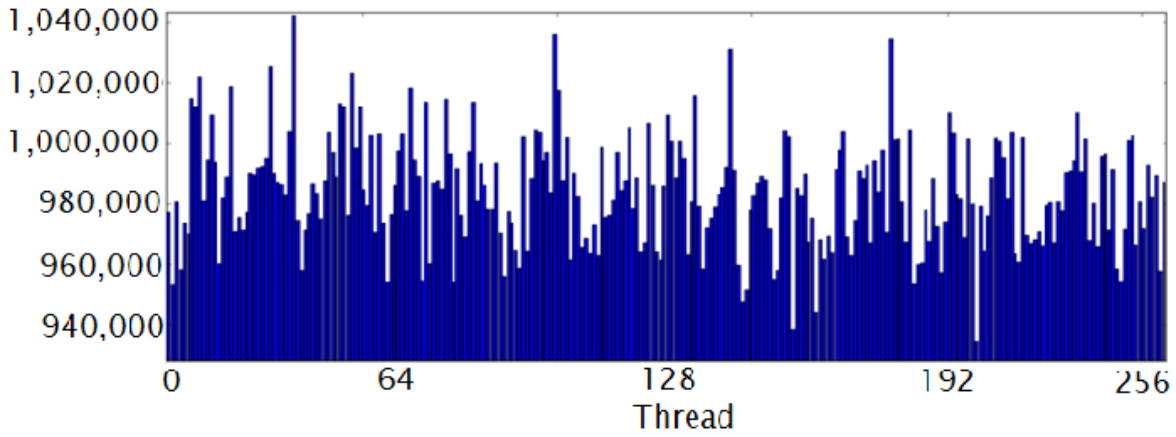


Figure 65: Load imbalance of dynamic instructions for CUDA SDK Mandelbrot application.

```
};
```

Listing 7.3: TraceEvent interface.

An example trace generator appears in Listing 7.4 which counts the number of dynamic instructions executed by each thread. This trace generator instance overrides the `event()` method and counts the number of active threads by examining the thread activity bitvector. Results for the CUDA SDK *Mandelbrot* application are presented in Figure 65. This example accumulates dynamic instructions across all CTAs.

```
/* Computes number of dynamic instructions for each thread  */
class ThreadLoadImbalance: public trace::TraceGenerator {
public:
  std::vector< size_t > dynamicInstructions;

  /* For each dynamic instruction, increment counters of each thread that executes it */
  virtual void event(const TraceEvent & event) {
    if (!dynamicInstructions.size()) {
      dynamicInstructions.resize(event.active.size(), 0);
    }

    for (int i = 0; i < event.active.size(); i++) {
      if (event.active[i]) {
        dynamicInstructions[i]++;
      }
    }
  }
};
```

Listing 7.4: This example TraceGenerator class counts dynamic instructions for each thread.

Applications. Detailed instruction traces have been used profile and characterize applications as described in Chapters 3 and 4. Application correctness and performance tuning have been developed as trace generators that ship with the trunk version of GPU Ocelot. These have been used to identify invalid memory accesses, detect race conditions, and profile kernels to identify hot regions as well as bottlenecks to global memory. Trace generators have also been developed to interact with the MACSIM [2] heterogeneous architecture simulator and have enabled research into analytic power and performance models [79, 80] for GPUs, research into exploiting heterogeneity [107, 112], and power optimizations for GPUs [61].

Correctness Checks. Precisely inspecting and verifying the architectural state of an abstract virtual machine executing PTX kernels presents numerous opportunities for ensuring program correctness. We have developed several tools to verify PTX kernels as they are executing. Each of the following is implemented using the `TraceGenerator` interface as described and illustrates the usefulness and flexibility of this design.

The *MemoryChecker* compares the set of memory addresses accessed during a load or store PTX instruction to a set of valid regions. Each device maintains a data structure of memory allocations that is queried for every address to assert it falls within a valid region in the appropriate address space. This catches exceptional events that would normally experience segmentation faults or inexplicable kernel crashes on actual devices. Relevant information is presented to the caller including thread ID, the offending address, and nearby allocations.

RaceDetector identifies unsynchronized access to shared memory by constructing a table of thread IDs which annotate every addressable location in `.shared`. The table is initialized with null thread identifiers. On store instructions to shared memory, the ID of the writing thread is written to the annotation table at the location specified by the shared address. Load instructions query the table and compare the last written ID to the thread ID issuing the instruction. If they match, or if the annotation table contains a null value, then the access is synchronized. If the consuming thread is not equal to the last writing thread, and the annotation table does not contain

a null identifier, then the *RaceDetector* has discovered unsynchronized data flow between different threads. If enabled, an exception containing address information of the offending thread is thrown.

IntegratedDebugger provides an interactive command line interface for executing PTX kernels one instruction at a time. This functionality is very similiar to facilities provided by other command line debuggers such as gdb. The trace generator's `event()` method enters into a loop waiting on commands. Users may define breakpoints, watchpoints, display the contents of the register file, and probe global memory locations. Unfortunately, lack of access to a symbol table prohibits interacting with CUDA-level variables.

Performance Tuning. Beyond verifying program correctness, precise accounting of the distribution of executed instructions and utilization of interconnect and memory bandwidth enables identifying bottlenecks and critical paths of compute kernels. A detailed explanation for how to use GPU Ocelot for performance tuning is available in [97].

7.3.2 PTX Emulator Performance

Functional simulation is critical to drive GPU simulators such as MACSIM [2] and GPGPU-Sim [9], and yet simulators are notoriously difficult to parallelize. GPU Ocelot's PTX emulator was implemented with performance and flexiblity of obtaining traces as primary design goals. As described earlier, the PTX intermediate representation is designed to be compact and flat to minimize indirection and maximize locality. Each PTX instruction is implemented by one or more handlers selected by instruction opcode and data type. A loop over all active threads executes the instruction for each before moving to the next. This replicates the SIMD-like execution model enabling accurate modeling of thread reconvergence on dynamic instruction counts. Moreover, this implementation leads to faster emulator performance with no decoding logic within the inner thread loop and predictable traversals of data structures modeling the register file, local, and shared memory.

Table 13 lists raw performance in thousands of simulated instructions per second (KIPS) of

Table 13: PTX Emulator performance (KIPS) executing kernels for CUDA workloads with and without online trace generators.

Application	No Trace Generators	with Memory Checking	and Race Detection
C-BinomialOptions	96.48	93.48	79.09
C-MersenneTwister	16.28	15.89	15.85
C-MonteCarlo	93.31	90.3	90.93
C-Nbody	75.16	74.73	73.88
C-ScalarProd	148.01	139.18	race
C-SobolQRNG	373.83	364.38	race
P-cutcp	72.38	68.5	race
P-fft	0.24	0.24	0.24
P-lbm	244.94	222.45	221.84
P-mm	155.6	148.15	140.01
P-mri-q	362.75	354.56	352.73
P-spmv	144.85	135.94	135.99

kernel execution on GPU Ocelot's PTX emulator backend. This excludes overheads introduced in implementing APIs or transferring data. Results are presented with no trace generators, with memory checking enabled for all load and store accesses, and with race detection for shared memory accesses. Evidently, each additional trace generator reduces instruction throughput. Applications making extensive accesses to global memory are particularly impacted by the memory checker. Kernels utilizing shared memory experience overheads from the race detector, and several examples such as CUDA's *RadixSort* include intentional race conditions among threads within the same warp. As mentioned, these are written to exploit the SIMD execution of warps which implicitly synchronize every thread within the same warp on every instruction. For these workloads, *RaceDetector* must be disabled.

7.3.3 Multicore CPU

The multicore CPU device provides a compilation path for efficient execution of CUDA workloads on multicore CPUs without emulation. Executing kernels requires mapping the CUDA execution model onto the hardware features available in mainstream CPUs and is the subject of Chapter 5.

Additionally, the PTX instruction set must be translated to the native instruction set architecture of the target processors, notably x86, x86-64, and ARM. Features from PTX not supported in the target ISA must be emulated, such as special mathematics functions including transcendental operators and texture sampling. Instructions relating to the execution model such as barrier synchronization, reductions, and lane voting denote semantic information used during execution model translation. This section provides additional detail describing ISA translation, implementation details in targeting the LLVM Intermediate Representation [105], and additional aspects of Ocelot's multicore CPU backend.

Translation to LLVM The Low Level Virtual Machine (LLVM) [105] is a maturing compiler infrastructure that maintains a strongly-typed program representation throughout compilation and link time. Multiple back-end code generators exist to translate LLVM IR to various popular instruction set architectures including x86 and x86-64. LLVM's IR itself includes explicit load and store instructions, integer and floating-point arithmetic, binary operators, and control flow operators. LLVM includes optimization passes that apply transformations to this intermediate form including well-known compiler optimizations such as common subexpression elimination, dead code removal, and constant propagation. By translating from one intermediate form to LLVM's IR and then leveraging LLVM's existing optimization and code generation components, a developer may construct a complete path to native execution on popular CPU architectures. Table 14 summarizes procedures for translating each class of PTX instruction.

In Figure 66, an example kernel expressed in CUDA is first compiled by NVIDIA's CUDA compiler (nvcc) producing a PTX representation which is then translated by Ocelot's LLVM Translation framework yielding the kernel on the right side of the figure. This is an LLVM representation of the kernel that could be executed by one host thread for each thread in the CTA and correctly execute the kernel. Note the runtime support structure providing block and thread ID.

```
//
// CUDA
//
extern "C" __global__
void simple(int *A) {
  int i = threadIdx.x +
    blockIdx.x * blockDim.x;
  A[i] = i;
}
```

Listing 7.5: CUDA kernel.

```
// PTX
.entry simple(
  .param .u64 _param_0) {

  .reg .s32   %r<6>;
  .reg .s64   %rl<5>;

  ld.param.u64 %rl1, [_param_0];
  cvta.to.global.u64 %rl2, %rl1;
  mov.u32     %r1, %ntid.x;
  mov.u32     %r2, %ctaid.x;
  mov.u32     %r3, %tid.x;
  mad.lo.s32  %r4, %r1, %r2, %r3;
  mul.wide.s32   %rl3, %r4, 4;
  add.s64   %rl4, %rl2, %rl3;
  st.global.u32   [%rl4], %r4;
  ret;
}
```

Listing 7.6: Compiled to PTX.

Figure 66: Compiling CUDA to PTX via nvcc. An LLVM translation of this kernel appears in Listing 7.7.

```
// LLVM
//
define internal void @_subkernel_simple_1_opt3_ws1(
  %LLVMContext* %__ctaContext) nounwind align 1 {

  %0 = getelementptr %LLVMContext* %__ctaContext, i64 0, i32 1, i32 0
  %blockDim.x.t0 = load i32* %0, align 4
  %1 = getelementptr %LLVMContext* %__ctaContext, i64 0, i32 2, i32 0
  %blockId.x.t0 = load i32* %1, align 4
  %argumentPtrPtr.t0 = getelementptr %LLVMContext* %__ctaContext, i64 0, i32 8
  %argumentPtr.t0 = load i8** %argumentPtrPtr.t0, align 8
  %ptrThreadCount = getelementptr %LLVMContext* %__ctaContext, i64 0, i32 12
  %rt9 = mul i32 %blockId.x.t0, %blockDim.x.t0

  %2 = bitcast i8* %argumentPtr.t0 to i64*
  %lsr.iv3 = bitcast %LLVMContext* %__ctaContext to i32*
  %threadId.x.t0 = load i32* %lsr.iv3, align 4
  %r0 = load i64* %2, align 8
  %r5 = add i32 %threadId.x.t0, %rt9
  %rt10 = sext i32 %r5 to i64
  %r6 = shl nsw i64 %rt10, 2
  %r7 = add i64 %r0, %r6
  %rt11 = inttoptr i64 %r7 to i32*
  store i32 %r5, i32* %rt11, align 4

  ret void
}
```

Listing 7.7: Translating the code in Listing 7.6 from the PTX instruction set to LLVM IR.

PTX defines several built-in registers which may be read to determine grid dimensions, CTA size, CTA identity, and thread identity. Additional registers enable the reading of performance monitors, warp size, and a high-performance cycle counter. To accommodate these, the runtime maintains light-weight thread context information that is updated on context switches.

Table 14: PTX to LLVM ISA translation rules.

PTX Statement	Target	Comment
Variable declaration	Alloca or value	Value with address taken must be allocated in thread-local memory
Arithmetic	LLVM IR	Nearly one-to-one correspondence
Special registers	`LLVMContext`	Context object
Transcendental operators	LLVM intrinsics	Lowered onto target ISA
Atomics	LLVM intrinsics	Lowered onto target ISA
Texture sampling	Ocelot Runtime	Software emulation
Barrier	context-switch	Compiler-inserted yield handler
Reductions	context-switch	Compiler-inserted yield handler
Call	context-switch	Compiler-inserted yield handler

LLVM functions are expressed in strict single-assignment form with explicit use-def chains for each value. PTX, on the other hand, expresses computations in terms of source and destination registers. To express PTX in LLVM instructions, Ocelot first performs control- and data-flow analysis to construct def-use chains for each value and rename registers accordingly. *phi* functions joining values are placed at the dominance frontiers of generating instructions to complete the full-SSA form expression [8,119] of the function. Recall that *phi* functions are not actual computations but placeholders to identify the several expressions that may produce a particular value depending on control paths taken [8]. This step is performed on the PTX representation prior to translation to compute precise data-flow and thereby enumerate live values. Additionally, precise data-flow analysis facilitates compiler analyses and optimization opportunities such as live range splitting, register allocation, partial redundancy elimination, partial dead code elimination, and many others. See [119] for a detailed treatment of classical compiler optimizations.

LLVM assumes strong type checking and requires explicit conversions between types. PTX, on the other hand, expresses instructions in terms of types but implicitly casts source and destination registers. During translation, explicit conversions are made using LLVM's bitcast instruction. Additionally, while PTX supports each of the IEEE 754 rounding modes (to nearest, to infinity, to -infinity, to zero), LLVM only supports rounding to nearest int. For those instructions specifying floating-point rounding modes, an additional instruction is issued to add or subtract 0.5 to the result prior to rounding in the case of +infinity and -infinity. In the case of round to zero, 0.5 is conditionally added or subtracted from the result depending on whether the number is greater than or or less than zero.

PTX defines several built-in registers which may be read to determine grid dimensions, CTA size, CTA identity, and thread identity. Additional registers enable the reading of performance monitors, warp size, and a high-performance cycle counter. To accommodate these, the runtime maintains light-weight thread context information that is updated on context switches.

CTA Runtime Support. When an application launches a kernel, a multi-threaded runtime layer launches as many worker threads as there are hardware threads available in addition to a context data structure per thread. This context consists of a block of shared memory, a block of local memory used for register spills, and special registers. Worker threads then iterate over the blocks of the kernel grid and each executes block as a CTA. The execution model permits any ordering of CTAs and any mapping to concurrent worker threads.

PTX defines several instructions which require special handling by the runtime. PTX compute capability 1.1 introduces atomic global memory operators which implement primitive transactional operations such as exchange, compare and swap, add, increment, and max, to name a few. Because global memory is inconsistent until a kernel terminates, we faced several options for implementing atomic accesses. The simplest option places a global lock around global memory. GPUs typically

provide hardware support for texture sampling and filtering. PTX defines a texture sampling instruction which samples a bound texture and optionally performs interpolation depending on the GPU driver state. As CPUs do not provide hardware support for texture sampling, Ocelot performs nearest and bilinear interpolation in software by translating PTX instructions into function calls which examine internal Ocelot data structures to identify mapped textures, compute addresses of referenced samples, and interpolate accordingly. Additionally, PTX includes several other instructions that do not have trivial mappings to LLVM instructions. These include transcendental operators such as *cos* and *sin* as well as parallel reduction. These too are implemented by calls into the Ocelot runtime which in turn calls C standard library functions in the case of the floating-point transcendentals.

Reductions and implicitly synchronized accesses to shared memory present several challenges for the multicore CPU backend, whose warp size may be as small as one thread. Most PTX kernels do not query the machine warp size before relying on a fixed size (32 threads) for program correctness. Consequently, these applications are not portable and fail on platforms in which the warp size is fewer than 32 threads such as GPU Ocelot's multicore background. Warp-wide reductions such as PTX's `red` and `vote` instructions are not implemented, though partial coverage may be possible by placing explicit barriers around them via compiler transformations and inserting code to perform the reduction in software. This is not guaranteed to be correct in all cases and solves a problem related to prematurely optimizing PTX kernels in ways that are ambiguously treated by the PTX specification. Implicit synchronization remains an effective optimization for obtaining peak instruction throughput in compute-bound kernels on GPUs but violates the SIMT definition of CUDA and OpenCL.

Performance of Generated Code. The multicore backends implemented with LLVM apply scalar optimizations derived from traditional compiler research available from the LLVM project. The LLVM code generator is particularly aggressive in selecting optimal instruction sequences, at the expense of code generation runtimes.

Table 15: Normalized execution time of different LLVM passes compared to the baseline with no optimization.

Application	ConstantPropagation	DeadInstElimination	DeadCodeElimination	DeadStoreElimination	AggressiveDCE	InductionVariableSimplify	InstructionCombining	LoopInvariantMotion	LoopStrengthReduce	LoopUnswitch	LoopUnroll	LoopRotate	TailDuplication	JumpThreading	CFGSimplification	BlockPlacement	GlobalValueNumber	GEPSpliter	SCCVN	All Optimizations
CP	1.03	1.04	1.01	1.03	1.02	1.01	1.01	1	1.02	1.04	1.01	1.03	1.02	1.02	.96	1.04	1	.99	1.02	.99
MRI-FHD	.89	1.14	.92	.92	.95	.9	.77	1	1	.92	1.02	.88	.9	.81	.97	.94	.86	.88	.74	.99
MRI-Q	1.04	.92	.88	1.06	1.03	.9	1.03	.88	.98	1.01	.9	1.18	1.01	1	1.01	.92	1	.97	.93	.99
PNS	1.02	.97	1.01	.98	1.02	.97	1.02	1.02	.98	1	1.02	1	1	.95	1	1.01	.97	.98	1.02	1
RPES	1	1.05	1.01	1	.99	1.01	1.03	1	1	1	1.01	1.02	1	1.02	1.03	1	1	.99	.91	.98
SAD	1.15	1.21	1.18	1.16	1.12	1.47	1.18	1.2	1.13	1.12	1.18	1.15	1.15	1.16	1.1	1.53	1.32	1.12	1.2	.98
TPACF	.84	.86	.76	.88	.87	.89	.89	.89	.85	.86	.8	.8	.78	.84	.87	.83	.88	.84	.9	.85
Average	.98	.99	.95	.98	.97	.98	.99	.98	.96	.98	.96	.96	.95	.97	.98	.98	.97	.96	.95	.97

Table 15 presents speedup. These results also include normalized runtimes with all optimizations enabled. No overheads are included in this experiment, only time spent executing translated code is counted. On average, LLVM optimizatins improve execution time by 1% to 5%. Some applications, such as *TPACF*, universally benefit from optimization while others, such as *SAD*, uniformly slow down. It is also clear that certain optimizations are more suitable to specific applications. For example, *MRI-FHD* benefits the most from instruction combining, which negatively impacts the performance of *PNSP*. In several applications, optimizations such as *ConstantPropagation* and *InductionVariableSimplify* achieve faster execution individually than when all optimization passes are applied. Yet, this relationship does not hold for every application. Overall these per-thread optimizations yield relatively minor improvements in execution time, indicating that the optimizations performed statically by NVCC, during code generation by LLVM, and dynamically by the CPU instruction scheduling logic are already highly tuned. This motivates shifting focus away from optimizations within single threads to address problems related to the thread hierarchy. This includes improving memory access patterns via better thread and CTA schedules and eliminating redundancy via interthread analysis. These optimizations in particular were directly addressed in Chapter 5 and Chapter 6 of this thesis.

7.3.4 NVIDIA GPU

The NVIDIA GPU backend enables execution of PTX kernels and CUDA workloads on NVIDIA GPUs via interaction with the CUDA Driver API. This device backend is meant to enable experiments regarding compilation for GPUs such as reallocating registers, applying structural program transformations, profile-guided optimizations, and managed execution of heterogeneous workloads. This device backend interacts with attached NVIDIA GPUs via the CUDA Driver Application Programming Interface. This is a low-level API on which NVIDIA's own CUDA Runtime API is implemented as well as several existing commercial and research products such as NVIDIA's OptiX [128]. The driver API provides a device-oriented approach to registering PTX modules, obtaining handles referencing kernels and global variables, managing memory, and initiating kernel execution. The CUDA Driver API presents more flexible access to managing several devices that may be present and features a device context stack for isolating interacting with CUDA devices to enable better composability. For example, a CUDA application may invoke a library that happens to be implemented using CUDA. The client library may push its own device context onto the active thread's CUDA device context stack.

GPU Ocelot makes use of this more powerful API in several ways. First, Ocelot enables threads to change their active device at any point within execution of the host program (though this call blocks until all kernels have executed). This contrasts with the CUDA Runtime API's `cudaSetDevice()` API call which may only be called once per active host thread. Secondly, GPU Ocelot may imperatively register and utilize PTX modules as the application is executing. The CUDA Runtime API does not present a documented methodology for doing this and assumes a static compliation model. Consequently, GPU Ocelot provides interfaces that make it significantly easier to interact with CUDA applications at the PTX level, to develop and optimize kernels while enjoying the benefits of a higher-level API like the CUDA Runtime API, and to dynamically construct and execute PTX modules on the fly.

Instrumentation. To explore applications of dynamic instrumentation, GPU Ocelot was extended to present an interface for implementing PTX instrumentation tools and provides an externally visible API for attaching instrumentation passes to Ocelot before and during the execution of GPU compute applications. As described in previous work [95], Ocelot replaces NVIDIA's CUDA Runtime API library (`libcudart` on Mac and Linux, `cudart.dll` on Windows) during the link step when CUDA applications are compiled. To insert third party instrumentation procedures, applications can be modified to explicitly add and remove instrumentors between kernel launches of the program via Ocelot's `add` and `remove` APIs. Alternatively, instrumentation tools built as an additional library and linked with the application may add themselves when the library is initialized. This approach means application sources do not need to be modified or recompiled. Farooqui et al. provide a complete discussion and evaluation of Ocelot's PTX instrumentation in [57]. Extensions to this work including the addition of a C-to-PTX compilation toolchain is available in [56].

The instrumentation tools themselves are C++ classes that consist of two logical components: (1) an instrumentor class derived from the abstract base class `PTXInstrumentor`, and (2) an instrumentation pass class derived from Ocelot's `Pass` abstract class. The instrumentor is responsible for performing any static analysis necessary for the instrumentation, constructing instrumentation-related data structures, instantiating a PTX transformation pass, extracting instrumentation results, and cleaning up resources. The PTX pass applies transformations to PTX modules which are presented to it via Ocelot's PTX Intermediate Representation (IR).

Certain instrumentations may require inspection of the kernel's CFG to obtain necessary information required by the CUDA Runtime API to properly allocate resources on the device. In general, any actions that must be performed prior to allocating resources on the device are encapsulated in the `analyze()` method. For our basic block execution count instrumentation, we obtain the CFG of each kernel to determine the total number of basic blocks.

Before launching a kernel, memory on the device must be allocated and initialized to store

the instrumentation results. Ocelot calls each registered instrumentation pass's `initialize()` method which may allocate memory and transfer data to and from the selected device. After the kernel has been launched, each instrumentor's `finalize()` method is invoked to free up allocated resources and extract instrumentation results into an instance of `KernelProfile`. The `KernelProfile` class outputs results either to a file or database, or it may channel instrumentation results to other components or applications that link with Ocelot.

The `BasicBlockInstrumentationPass` constructs a matrix of counters with one row per basic block in the executed kernel and one column per dynamic PTX thread. By assigning one basic block counter per thread, the instrumentation avoids contention to global memory that would be experienced if each thread performed atomic increments to the same block counter. Instrumentation code added via a `BasicBlockPass` loads a pointer to the counter matrix from a global variable. The instrumentation pass then adds PTX instructions to each basic block that compute that thread's counter index and increments the associated counter using non-atomic loads and stores. Counters of the same block for consecutive threads are arranged in consecutive order in global memory to ensure accesses are coalesced and guaranteed to hit the same L1 cache line. At runtime, this instrumentation pass allocates the counter matrix sized according to the kernel's configured block size.

Hot Region Detection. To determine the most frequently executed basic blocks within the kernel, we use our `BasicBlockInstrumentationPass`. Figure 67 is a heat map visualizing the results from this experiment for the Scan application from the CUDA SDK. Basic blocks are colored in intensity in proportion to the number of threads that have entered them. The hottest region consists of blocks `BB_001_007`, `BB_001_008`, `BB_001_009` corresponding to this kernel's inner loop. This metric captures architecture-independent behavior specified by the application. A similar instruction trace analysis offered by Ocelot's PTX emulator provides the same information but at the cost of emulation. By instrumenting native PTX and executing the kernels natively on a Fermi-class NVIDIA GTX480 [125], a speedup of approximately 1000x over the emulator was

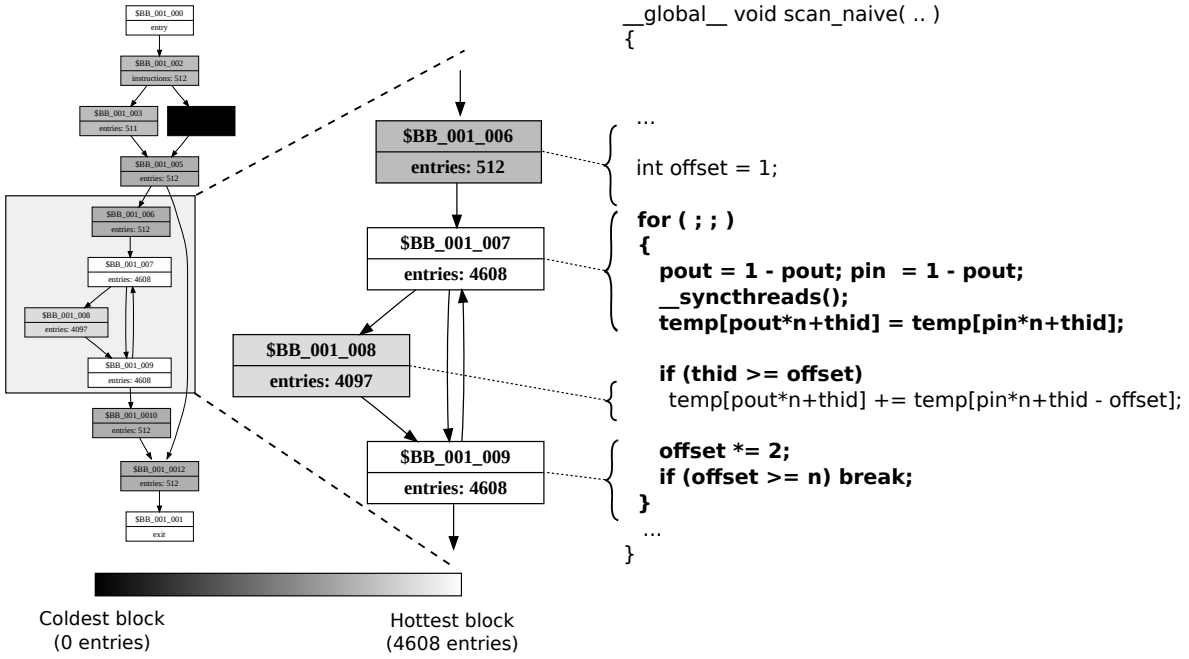


Figure 67: Hot region visualization of CUDA SDK Scan application profiled during native GPU execution. Each block presents a count of the number of times a thread entered the basic block and is color coded to indicate computational intensity. The magnified portion of the control-flow graph illustrates a loop, the dominant computation in the kernel.

achieved. This is an example of workload characterization accelerated by GPUs.

Overhead of Instrumentation. The basic block execution count instrumentation contributes a per-block overhead in terms of memory bandwidth and computation. Blocks in the hottest region make numerous accesses when incrementing their respective per-thread counters and displace some cache lines from the L1 and L2 caches. Clock cycle count instrumentation inserts instructions to read clock cycles at the beginning of the kernel and then to store the difference into a counter in global memory. All forms of instrumentation can be expected to perturb execution times in some way. This experiment measures runtimes of sample applications with and without each instrumentation pass. Slowdowns for selected applications from the CUDA SDK and Parboil appear in Figure 5. These applications cover a spectrum of structural properties related to basic block instrumentation. Properties include number of operations per basic block, number of kernels launched, and whether they are memory- or compute-bound.

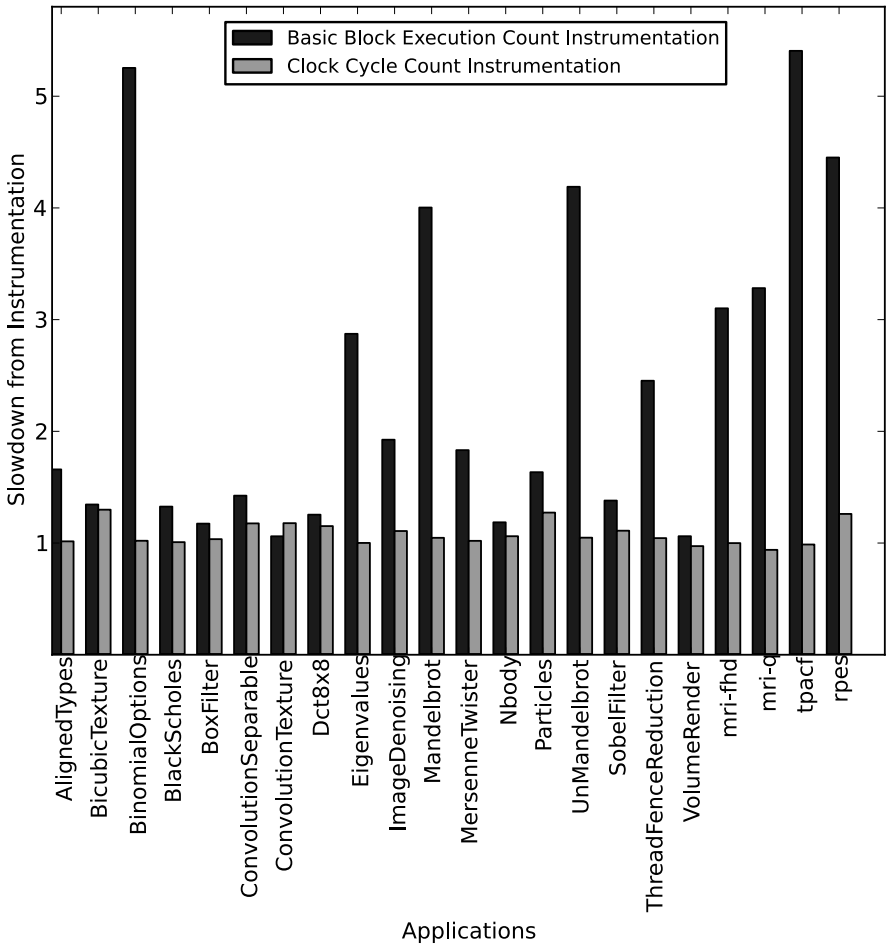


Figure 68: Slowdowns of selected applications due to `BasicBlockInstrumentor` and `ClockCycleCountInstrumentor`.

Characterization of JIT Compilation Overheads. Dynamic binary instrumentation invokes a compilation step as the program is running. Application runtime is impacted both by the overheads associated with executing instrumentation code when it is encountered and also by the process of inserting the instrumentation itself. Dynamically instrumented CUDA programs require an additional just-in-time compilation step to translate from PTX to the native GPU instruction set, but applications are typically written with long-running kernels in mind. In this experiment, we attempt to characterize overheads in each step of Ocelot's compilation pipeline from parsing large PTX modules, performing static analysis, executing PTX-to-PTX transformations, JIT compiling

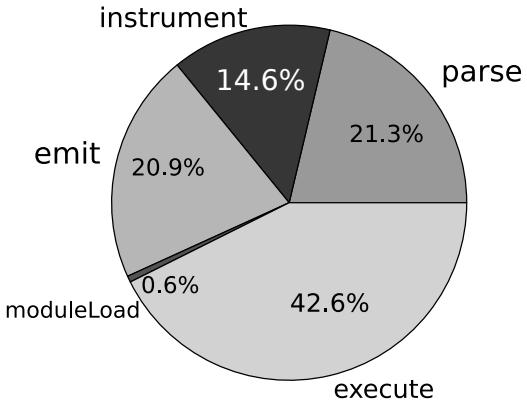


Figure 69: Overheads in compiling and executing the MRI-FHD application from the Parboil benchmark suite. Instrumentation occupies 14.6% of total kernel runtime including compilation.

via the CUDA Driver API, and executing on the GPU.

Figure 69 presents the dynamic compilation overheads in compiling and executing the Parboil application *mri-fhd* and instrumenting it with the basic block counters. This application consists of a single PTX module of moderate size (2,916 lines of PTX). The figure shows the relative time spent performing the instrumentation passes, 14.6%, is less than both the times to parse the PTX module and to re-emit it for loading by the CUDA Driver API, steps that would be needed without adding instrumentation. Online use of instrumentation would not need to perform the parse step more than once. Results indicate there would be less than a 2x slowdown if kernels were instrumented and re-emitted with each invocation, a slowdown which would decrease for longer running kernels.

7.3.5 AMD GPU

The AMD GPU backend contributed by Dominguez et al. [51] demonstrates the feasibility of targeting other mainstream GPU architectures with the PTX execution model. This work approached three main challenges: (1) unstructured control flow, (2) accommodating a more structured memory hierarchy, and (3) targeting combined SIMD-VLIW architectures. At the time of this work,

AMD GPUs do not feature arbitrary control instructions such as conditional branches. Rather, nested `if-else` and `do-while` instructions create structured control-flow with explicit reconverge locations. Techniques for transforming unstructured control flow graphs to structured control trees are described in [51, 119, 154]. Secondly, AMD GPUs expose multiple paths to off-chip memory to software, depending on whether accesses are read-only or guaranteed to be aligned. This places additional burden on code generation to select the appropriate path. Thirdly, AMD GPU instructions contain multiple operations, as this line of GPU architecture are both SIMD and five-way VLIW. Without utilizing static instruction scheduling techniques such as modulo scheduling, reaching peak instruction throughputs is impossible. Nevertheless, later architectures such as NVIDIA's Kepler [127] indicate statically scheduled instruction windows and VLIW processor architectures are energy efficient and programmable given strong compiler support.

7.4 Extending the Device Interface

This section describes extensions to the device interface enabled by the modular and layered design of GPU Ocelot.

7.4.1 Device Switching

Ocelot tracks all device state - memory allocations, textures, and PTX modules - and may therefore rematerialize device state in other contexts. We apply device state serialization in two contexts: (1) device switching and the (2) kernel extractor utility. Unlike most currently available implementations of OpenCL, an ostensibly vendor neutral programming model and alternative to CUDA, GPU Ocelot's selected device may be changed dynamically in response to an externally specified device context switch API. This serializes device state, copies it to the destination device's address space, and reconstructs all allocations. GPU Ocelot's JIT compilation framework enables running kernels on the new devices as described extensively in this work.

State serialization itself is one of the challenges addressed in checkpointing, namely the problem of restarting the execution of an application when virtual addresses have changed. If all pointers reference base addresses of memory allocations, and all other references are obtained by adding offsets to base addresses, then rematerializing data structures is possible simply by mapping base pointers from the source address space to the pointers of new allocations in the destination and then copying each allocation to the destination. Alternatively, if pointers are permitted to reference any location and may be embedded in data structures, then the general case requires application-specific traversing of each data structure to update them. Analysis of the high-level program representation may facilitate the automation of such a traversal, but this work is focused on lower-level program representations in which type information, particularly to pointers, is not easily available. This problem is discussed in greater detail in related work in the context of checkpointing for reliability [38, 63]. OpenCL enforces the constraint of forbidding the use of pointer-valued variables at the language level. Only kernel paremters may be pointer-valued which are explicit and easily filtered before a kernel is executed. Data structures must rely on position-independent offsets or some other form of references to store references. CUDA defines no such restriction, enabling much more flexibility but greatly compounding problems related to points-to analysis.

GPU Ocelot defines an API, `ocelot::contextSwitch()`, which initiates data movement between device address spaces and returns a pointer between all allocations on the old device to corresponding allocations on the new device. While not completely transparent, this enables the application programmer to use this pointer map however it may be needed to update data structures. Thus, GPU Ocelot is a truely heterogeneous abstraction layer decoupling computations from actual devices and providing procedures for changing devices dynamically.

7.4.2 Kernel Extractor

GPU Ocelot reimplements the CUDA Runtime API enabling applications to run transparently if they are implemented with CUDA. However, several GPU compute applications do not use

the CUDA Runtime API directly. These include NVIDIA's OptiX [128] raytracer infrastructure. Other applications are available in binary-only form such as CUBLAS [122] and third-party GPU-accelerated libraries. In many cases, these applications have unique behaviors that would be helpful to have available for architectural studies or performance evaluations.

Thus, GPU Ocelot's *Kernel Extractor* utility was written to leverage elements of the PTX IR and data structures to describe device state. This is a light-weight wrapper around the CUDA Driver API that is loaded transparently by CUDA applications executing otherwise normally on their configured GPU device. Before and after each kernel launch, the complete reachable state space of the GPU is serialized and saved to disk. Afterwards, this serialized device state may be loaded and replayed by a GPU Ocelot application. This approach was used to complete a study on thread reconvergence [47] in which several applications of interest could not be run through GPU Ocelot directly, including OptiX.

7.4.3 Remote Device

GPU Ocelot presents a convenient way to implement a custom Remote Procedure Call (RPC) layer and enable multi-GPU programming to extend to multiple nodes. CUDA applications are already tolerant of high-latency API calls, many of which are non-blocking, so the additional latency of transmitting an RPC message to a remote node does not significantly degrade application runtimes. Kernel execution itself takes place on GPUs installed in the remote node without perturbation. `cudaMemcpy()` calls require transmitting larger blocks of data and is limited by network bandwidth. This work satisfies a similar purpose as the rCUDA [129] project, and the RPC implementation is similar to efforts to virtualize GPUs within clusters [118].

7.5 Conclusion

The GPU Ocelot infrastructure was made available as an open source project in June 2009 and has since been used as a compiler framework and simulator for numerous research efforts at

the Georgia Institute of Technology and elsewhere. Its complete intermediate representation of PTX, production-quality parser and emitter, and compilation analysis and transformation framework have enabled numerous research results developed outside of this research group. The PTX emulator has produced results for numerous architecture and characterization studies both in this work and beyond. The multicore backend developed for this research has been cited as an efficient solution to explore data-parallel computing on architectures beyond GPUs. GPU Ocelot is a maintained and tested codebase and has been the subject of several invited talks and tutorials.

CHAPTER VIII

CONCLUSION

This chapter presents final conclusions drawn for development of heterogeneous compilation frameworks.

8.1 Summary

This dissertation addresses the portability and performance challenges of adopting heterogeneous computing platforms which emphasize parallelism. We define metrics for characterizing bulk-synchronous workloads, modeling performance, targeting parallel execution models to multiple classes of processor architectures, and overcoming dynamic compilation overheads. To evaluate the viability and utility of a heterogeneous dynamic compiler, this work has developed a candidate implementation targeting mainstream commodity parallel processors. By supporting NVIDIA's CUDA, a well-known and widely adopted programming language and API, we were able to evaluate techniques on real-world workloads with varied and unbiased characteristics.

8.2 Contributions

The contributions of this dissertation may be summarized as follows:

- Characterization of kernel-oriented workloads
- Predictive performance modeling via statistical methods
- Execution model translation to target commodity CPUs with SIMD instruction sets
- Region-based compilation to reduce overheads and improve scheduling opportunities

This dissertation defines several metrics to characterize data-parallel programming models and workloads. In particular, these metrics indicate high correlation between the control paths and data access patterns of multiple threads within the same kernel. This follows from the specification of the execution model and typical hardware realizations which emphasize uniformity across threads to maximize execution efficiency. Nevertheless, the programming model does permit divergence, and some divergence is present in nearly all of the benchmark applications that were studied. This presents several implications for correctness and optimization of software realizations of the execution model, particularly when constrained by hardware limitations in other processor architectures.

This dissertation demonstrates execution model portability by targeting vector processors with different ISAs, notably mainstream CPUs with SIMD instruction set extensions. This work reveals challenges related to lack of hardware support and develops the concept of specializing for different vector widths as a method to accommodate divergent control flow. Evaluations on real world platforms achieved speedups approaching the vector width of the processor. Lack of support for predication in the target ISA motivated the decision to rely on compiler specialization rather than control-flow restructuring, a design choice which was satisfiable by dynamic compilation. This thesis also describes several optimizations that are enabled when the compiler is able to reason about the execution of several threads such as eliminating invariant expressions as well as vectorizing affine memory accesses.

To reduce startup overheads, this dissertation explores a region-based compilation scheme that partitions kernels prior to translating them. Control edges between partitions are treated as context switches and enable more complex thread schedules which maximize instruction reuse and exploit data locality. This is applied to specialization and vectorization and demonstrates a reduction in compiled code size for most applications as well as a reduction in cache miss rates due to better scheduling of threads.

8.3 Future Work

While targeting currently available multicore CPUs has practical applications, roadmaps from Intel, AMD, ARM, and others indicate rising levels of on-die heterogeneity and greater reliance on parallelism. Intel and AMD have extended the vector width of their processors from four to eight with the inclusion of AVX. As code generators become more mature, directly applying the implementations developed in GPU Ocelot is likely to yield speedups.

Future Many-core Processors

Further out, Intel's Knight's Corner promises even wider vector widths and much greater emphasis on optimizing throughput-oriented workloads. Hardware support for predication and scatter-gather instructions may mitigate some of the control-flow intolerance of current implementations but may do more to reveal the utility of permitting the compiler to statically interleave the instruction streams of multiple logical threads. We expect the concepts developed in this thesis to be particularly applicable to massively parallel CPU-like vector processors, particularly as data-parallel execution models become more prevalent ways to program them.

This work extends naturally to tightly coupled GPU-CPU combinations, most notably the AMD Fusion [21] architecture which devotes substantial die area to *Auxilliary Processing Units* which are organized similarly to GPU cores. Support for efficiently spawning threads and kernels for short data-parallel computations and controlling the granularity of concurrency through software would enable adoption using other programming models that are less explicitly parallel. The capacity to nest parallelism promises to improve the composability of kernels. Applying the techniques described on these types of processors may yield interesting new capabilities such as executing subkernels concurrently on both GPU and CPU architectures.

Interprocedural Kernel Transformations

Dynamic compilation lends itself well to interprocedural analysis and automatically tuning kernel parameters based on application call graphs. Wu et al. [155] propose *kernel fusion* as a method

to compose kernels by combining function bodies of several kernels to reduce data movement. Merrill [116] and Cohen [30] use template metaprogramming to construct specialized, policy-based kernels optimized for particular launch configurations and workloads. In each of these cases, kernel composition is achieved statically at compile time. This forbids using powerful execution model features such as dynamic binding and polymorphism, each of which limit interprocedural analysis.

Profiling application execution and fusing or tuning kernels at runtime would thus yield at least the same performance improvement as static efforts. Profile-driven low-level optimizations on GPU kernels executing on GPUs has not been thoroughly explored, mainly due to vendor opaqueness and lack of a suitable dynamic intermediate representation on which to mutate kernels. Data compaction transformations have achieved some success at improving performance of applications [159]. As irregular workloads that are difficult to optimize for in the general case become more common targets for dara-parallel processors like GPUs, performance tuning in response to program behavior, particularly memory and communication patterns, may become more critical drivers of optimizations.

REFERENCES

[1] "Openmp application programming interface," May 2008.

[2] "Macsim: Simulator for heterogeneous architecture," February 2012. `http://code.google.com/p/macsim/`.

[3] "Scipy: Scientific tools for python," July 2012. http://www.scipy.org/.

[4] ADL-TABATABAI, A.-R., CIERNIAK, M., LUEH, G.-Y., PARIKH, V. M., and STICHNOTH, J. M., "Fast, effective code generation in a just-in-time java compiler," *SIGPLAN Not.*, vol. 33, pp. 280–290, May 1998.

[5] ADVE, S. V., BURGER, D., EIGENMANN, R., RAWSTHORNE, A., SMITH, M. D., GEBOTYS, C. H., KANDEMIR, M. T., LILJA, D. J., CHOUDHARY, A. N., FANG, J. Z., and YEW, P.-C., "Changing interaction of compiler and architecture," *Computer*, vol. 30, pp. 51–58, December 1997.

[6] ALEEN, F. and CLARK, N., "Commutativity analysis for software parallelization: letting program transformations see the big picture.," in *ASPLOS'09*, pp. 241–252, 2009.

[7] ANASUA, B. and MANOJ, F., "A general compiler framework for speculative multithreading," in *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, (New York, NY, USA), pp. 99–108, ACM, 2002.

[8] AYCOCK, J. and HORSPOOL, N., "Simple generation of static single-assignment form," 2000.

[9] BAKHODA, A., YUAN, G., FUNG, W. W. L., WONG, H., and AAMODT, T. M., "Analyzing cuda workloads using a detailed gpu simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, (Boston, MA, USA), April 2009.

[10] BALA, V., DUESTERWALD, E., and BANERJIA, S., "Dynamo: a transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, (New York, NY, USA), pp. 1–12, ACM, 2000.

[11] BARIK, R., ZHAO, J., and SARKAR, V., "Efficient selection of vector instructions using dynamic programming," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 201–212, IEEE Computer Society, 2010.

[12] BIENIA, C. and LI, K., "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[13] BINDEL, D., DEMMEL, J., KAHAN, W., and MARQUES, O., *On computing Givens rotations reliably and efficiently*. ACM Trans. Math. Softw., New York, NY, USA, 2002.

[14] BISCHOF, C. H. and LOAN, C. V., *The WY Representation for Products of Householder Matrices*. Cornell University, Ithaca, NY, USA, 1985.

[15] BLELLOCH, G. E., *Vector models for data-parallel computing*. Cambridge, MA, USA: MIT Press, 1990.

[16] BONDHUGULA, U., HARTONO, A., RAMANUJAM, J., and SADAYAPPAN, P., "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Not.*, vol. 43, pp. 101–113, June 2008.

[17] BOYLE, J. and DYKSTRA, R., "A method of finding projections onto the intersection of convex sets in hilbert spaces," vol. 37, pp. 28–47, 1986.

[18] BRIDGES, M., VACHHARAJANI, N., ZHANG, Y., JABLIN, T., and AUGUST, D., "Revisiting the sequential programming model for multi-core," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pp. 69 –84, dec. 2007.

[19] BRIGGS, P., COOPER, K. D., and TORCZON, L., "Rematerialization," in *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, (New York, NY, USA), pp. 311–321, ACM, 1992.

[20] BROWNE, S., DONGARRA, J., GARNER, N., LONDON, K., and MUCCI, P., "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Supercomputing, ACM/IEEE 2000 Conference*, p. 42, nov. 2000.

[21] BURGESS, B., COHEN, B., DENMAN, M., DUNDAS, J., KAPLAN, D., and RUPLEY, J., "Bobcat: Amd's low-power x86 processor," *IEEE Micro*, vol. 31, pp. 16–25, Mar. 2011.

[22] BURKE, M. G., CHOI, J.-D., FINK, S., GROVE, D., HIND, M., SARKAR, V., SERRANO, M. J., SREEDHAR, V. C., SRINIVASAN, H., and WHALEY, J., "The jalapeno dynamic optimizing compiler for java," in *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99, (New York, NY, USA), pp. 129–141, ACM, 1999.

[23] BUTLER, M., BARNES, L., SARMA, D., and GELINAS, B., "Bulldozer: An approach to multithreaded compute performance," *Micro, IEEE*, vol. 31, pp. 6 –15, march-april 2011.

[24] CATANZARO, B., GARLAND, M., and KEUTZER, K., "Copperhead: compiling an embedded data parallel language," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, (New York, NY, USA), pp. 47–56, ACM, 2011.

[25] CHAITIN, G., "Register allocation and spilling via graph coloring," *SIGPLAN Not.*, vol. 39, pp. 66–74, Apr. 2004.

[26] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H., and SKADRON, K., "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.*, October 2009.

[27] CHERNOFF, A., HERDEG, M., HOOKWAY, R., REEVE, C., RUBIN, N., TYE, T., YADAVALLI, S. B., and YATES, J., "Fx!32: A profile-directed binary translator," *IEEE Micro*, vol. 18, pp. 56–64, March 1998.

[28] CIFUENTES, C. and MALHOTRA, V., "Binary translation: Static, dynamic, retargetable?," in *In Proceedings of the International Conference on Software Maintenance (ICSM*, pp. 340–349, IEEE, 1996.

[29] CLARK, N., HORMATI, A., YEHIA, S., MAHLKE, S., and FLAUTNER, K., "Liquid simd: Abstracting simd hardware using lightweight dynamic mapping," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), pp. 216–227, IEEE Computer Society, 2007.

[30] COHEN, J., "Opencurrent," June 2010. http://code.google.com/p/opencurrent.

[31] COHN, R. and LOWNEY, P. G., "Feedback directed optimization in compaq's compilation tools for alpha," in *In Proc. 2nd Workshop on Feedback Directed Optimization*, pp. 3–12, 1999.

[32] COLLANGE, S., DEFOUR, D., and PARELLO, D., "Barra, a modular functional gpu simulator for gpgpu," Tech. Rep. hal-00359342, 2009.

[33] COLLANGE, S., DEFOUR, D., and ZHANG, Y., "Dynamic detection of uniform and affine vectors in gpgpu computations," Tech. Rep. hal-00396719, Universite de Perpignan, University of California Davis, June 2009.

[34] COLWELL, R. P., NIX, R. P., O'DONNELL, J. J., PAPWORTH, D. B., and RODMAN, P. K., "A vliw architecture for a trace scheduling compiler," in *In The 2nd International conference on Architectural Support for Programming Languages and Operating Systems*, pp. 180–192, 1987.

[35] COMERFORD, R., "How dec developed alpha," *Spectrum, IEEE*, vol. 29, pp. 26 –31, jul 1992.

[36] CONTRERAS, G. and MARTONOSI, M., "Characterizing and improving the performance of the intel threading building blocks runtime system," in *International Symposium on Workload Characterization (IISWC 2008)*, September 2008.

[37] COOK, J., COOK, J., and ALKOHLANI, W., "A statistical performance model of the opteron processor," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 75–80, Mar. 2011.

[38] CORNWELL, J. and KONGMUNVATTANA, A., "Efficient system-level remote checkpointing technique for blcr," in *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pp. 1002 –1007, april 2011.

[39] CORPORATION, A., "The arm cortex-a9 processors." white paper, ARM, September 2009.

[40] CORPORATION, T., "Tilera tile-64." http://www.tilera.com.

[41] COUTINHO, B., SAMPAIO, D., PEREIRA, F. M. Q., and MEIRA JR., W., "Divergence analysis and optimizations," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, (Washington, DC, USA), pp. 320–329, IEEE Computer Society, 2011.

[42] CRAEYNEST, K., JALEEL, A., ECKHOUT, L., NARVAEZ, P., and EMER, J., "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," June 2012.

[43] DALLY, W. and TOWLES, B., "Route packets, not wires: on-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*, pp. 684 – 689, 2001.

[44] DAVIDSON, A. and OWENS, J. D., "Register packing for cyclic reduction: a case study," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, (New York, NY, USA), pp. 4:1–4:6, ACM, 2011.

[45] DEHNERT, J., GRANT, B., BANNING, J., JOHNSON, R., KISTLER, T., KLAIBER, A., and MATTSON, J., "The transmeta code morphing trade; software: using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pp. 15 – 24, march 2003.

[46] DENNARD, R. H., GAENSSLEN, F. H., RIDEOUT, V., and BASSOUS, E., "Design of ion-implanted mosfets with very small phsical dimensions," in *IEEE Journal of Solid-State Circuits*, October 1974.

[47] DIAMOS, G., ASHBAUGH, B., MAIYURAN, S., KERR, A., WU, H., and YALAMANCHILI, S., "Simd re-convergence at thread frontiers," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 11, (New York, NY, USA), p. 477488, ACM, 2011.

[48] DIAMOS, G., KERR, A., and YALAMANCHILI, S., "Gpuocelot: A binary translation framework for ptx.," June 2009. `http://code.google.com/p/gpuocelot/`.

[49] DIAMOS, G., KERR, A., YALAMANCHILI, S., and CLARK, N., "Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 353–364, ACM, 2010.

[50] DIAMOS, G. and YALAMANCHILI, S., "Speculative execution on Multi-GPU systems," in *24th IEEE International Parallel & Distributed Processing Symposium*, (Atlanta, Georgia, USA), 4 2010.

[51] DOMÍNGUEZ, R., SCHAA, D., and KAELI, D., "Caracal: dynamic translation of runtime environments for gpus," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, (New York, NY, USA), pp. 5:1–5:7, ACM, 2011.

[52] EBCIOGLU, K. and ALTMAN, E., "Daisy: Dynamic compilation for 100% architectural compatibility," in *Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on*, pp. 26 –37, jun 1997.

[53] EECKHOUT, L., VANDIERENDONCK, H., and DE BOSSCHERE, K., "Designing computer architecture research workloads," *Computer*, vol. 36, pp. 65–71, Feb 2003.

[54] ESMAEILZADEH, H., BLEM, E., AMANT, R., SANKARALINGAM, K., and BURGER, D., "Dark silicon and the end of multicore scaling," in *IIEE International Symposium on Computer Architecture 2011*, ISCA 2011, (San Jose, CA), 2011.

[55] FARKAS, K. L., CHOW, P., JOUPPI, N. P., and VRANESIC, Z., "The multicluster architecture: Reducing cycle time through partitioning," pp. 149–159, 1997.

[56] FAROOQUI, N., KERR, A., EISENHAUER, G., SCHWAN, K., and YALAMANCHILI, S., "Lynx: A dynamic instrumentation system for data-parallel applications on gpgpu architectures," in *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pp. 58 –67, april 2012.

[57] FAROOQUI, N., KERR, A., DIAMOS, G., YALAMANCHILI, S., and SCHWAN, K., "A framework for dynamically instrumenting gpu compute applications within gpu ocelot," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, (New York, NY, USA), pp. 9:1–9:9, ACM, 2011.

[58] FISHER, J. A., "Very long instruction word architectures and the eli-512," *SIGARCH Comput. Archit. News*, vol. 11, pp. 140–150, June 1983.

[59] FUNG, W. W. L., SHAM, I., YUAN, G., and AAMODT, T. M., "Dynamic warp formation and scheduling for efficient gpu control flow," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 407–420, IEEE Computer Society, 2007.

[60] GALLAGHER, D. M., CHEN, W. Y., MAHLKE, S. A., GYLLENHAAL, J. C., and HWU, W.-M. W., "Dynamic memory disambiguation using the memory conflict buffer," *SIGPLAN Not.*, vol. 29, pp. 183–193, November 1994.

[61] GEBHART, M., JOHNSON, D. R., TARJAN, D., KECKLER, S. W., DALLY, W. J., LINDHOLM, E., and SKADRON, K., "A hierarchical thread scheduler and register file for energy-efficient throughput processors," *ACM Trans. Comput. Syst.*, vol. 30, pp. 8:1–8:38, Apr. 2012.

[62] GENBRUGGE, D. and EECKHOUT, L., "Chip multiprocessor design space exploration through statistical simulation," *Computers, IEEE Transactions on*, vol. 58, pp. 1668 –1681, dec. 2009.

[63] GIOIOSA, R., SANCHO, J., JIANG, S., and PETRINI, F., "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, p. 9, nov. 2005.

[64] GOLUB, G. and LOAN, C. V., *Matrix Computations*. Baltimore, MD.: Johns Hopkins University Press, third ed., 1996.

[65] GOSWAMI, N., SHANKAR, R., JOSHI, M., and LI, T., "Exploring gpgpu workloads: Characterization methodology, analysis and microarchitecture evaluation implications," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pp. 1 –10, dec. 2010.

[66] GROUP, K. O. W., *The OpenCL Specification*, December 2008.

[67] GROVER, V., LEE, S., and KERR, A., "Plang: Translating nvidia ptx language to llvm ir machine," 2009.

[68] GUMMARAJU, J., MORICHETTI, L., HOUSTON, M., SANDER, B., GASTER, B. R., and ZHENG, B., "Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 205–216, ACM, 2010.

[69] GUO, Z., ZHANG, E. Z., and SHEN, X., "Correctly treating synchronizations in compiling fine-grained spmd-threaded programs for cpu," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, (Washington, DC, USA), pp. 310–319, IEEE Computer Society, 2011.

[70] HAMEED, R., QADEER, W., WACHS, M., AZIZI, O., SOLOMATNIKOV, A., LEE, B. C., RICHARDSON, S., KOZYRAKIS, C., and HOROWITZ, M., "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, (New York, NY, USA), pp. 37–47, ACM, 2010.

[71] HAN, T. D. and ABDELRAHMAN, T. S., "Reducing branch divergence in gpu programs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, (New York, NY, USA), pp. 3:1–3:8, ACM, 2011.

[72] HANEY, R., MEUSE, T., KEPNER, J., and LEBAK, J., *HPEC Challenge Overview*. MIT Lincoln Laboratory, 2005.

[73] HANK, R. E., HWU, W.-M. W., and RAU, B. R., "Region-based compilation: introduction, motivation, and initial experience," *Int. J. Parallel Program.*, vol. 25, pp. 113–146, Apr. 1997.

[74] HANK, R. E., MAHLKE, S. A., BRINGMANN, R. A., GYLLENHAAL, J. C., and MEI W. HWU, W., "Superblock formation using static program analysis," in *in Proceedings of the 26th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-26*, pp. 247–255, 1993.

[75] HARRIS, M., SENGUPTA, S., and OWENS, J., *Parallel Prefix Sum (Scan) in CUDA*. Addison Wesley, 2007.

[76] HILLIS, W. D. and STEELE, JR., G. L., "Data parallel algorithms," *Commun. ACM*, vol. 29, pp. 1170–1183, December 1986.

[77] HOBEROCK, J. and BELL, N., "Thrust: A parallel template library," 2009. Version 1.2.

[78] HOFFMANN, H., "Stream Algorithms and Architecture," Master's thesis, Massachusetts Institute of Technology, June 2003.

[79] HONG, S. and KIM, H., "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 152–163, 2009.

[80] HONG, S. and KIM, H., "An integrated gpu power and performance model," in *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, (New York, NY, USA), pp. 280–289, ACM, 2010.

[81] IMPACT, "The parboil benchmark suite," 2007.

[82] INTEL, "Intel graphics media accelerator x3000," tech. rep., 2009.

[83] Intel Corp., *Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency*, March 2008.

[84] INTEL CORPORATION, *Intel 64 and IA-32 Architectures Optimization Reference Manual*. No. 248966-018 in Intel 64 and IA-32 Optimization Manaul, Intel Corporation, March 2009.

[85] ITRS, "International technology roadmap for semiconductors, 2010 update, 2011.." http://www.itrs.net.

[86] JANSSEN, C. L., ADALSTEINSSON, H., and KENNY, J. P., "Using simulation to design extremescale applications and architectures: programming model exploration," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 4–8, Mar. 2011.

[87] JOHNS, C. R. and BROKENSHIRE, D. A., "Introduction to the cell broadband engine architecture," *IBM J. Res. Dev.*, vol. 51, pp. 503–519, September 2007.

[88] JONES, T. M., O'BOYLE, M. F. P., ABELLA, J., GONZÁLEZ, A., and ERGIN, O., "Energy-efficient register caching with compiler assistance," *ACM Trans. Archit. Code Optim.*, vol. 6, pp. 13:1–13:23, October 2009.

[89] KANTER, D., "Intel's sandy bridge microarchitecture," tech. rep., Real World Technologies, 2010.

[90] KARRENBERG, R. and HACK, S., "Whole-function vectorization," in *Proceedings of the 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2011, 2011.

[91] KELM, J. H., JOHNSON, D. R., JOHNSON, M. R., CRAGO, N. C., TUOHY, W., MAHESRI, A., LUMETTA, S. S., FRANK, M. I., and PATEL, S. J., "Rigel: an architecture and scalable programming interface for a 1000-core accelerator," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, (New York, NY, USA), ACM, 2009.

[92] KERR, A., CAMPBELL, D., and RICHARDS, M., "Benchmarking gpus with hpec challenge," in *HPEC'08: High Performance Embedded Computing Workshop*, (Lexington, MA, USA), 2008.

[93] KERR, A., CAMPBELL, D., and RICHARDS, M., "Gpu vsipl: High-performance vsipl implementation for gpus," in *HPEC'08: High Performance Embedded Computing Workshop*, (Lexington, MA, USA), 2008.

[94] KERR, A., CAMPBELL, D., and RICHARDS, M., "Qr decomposition on gpus," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, (New York, NY, USA), pp. 71–78, ACM, 2009.

[95] KERR, A., DIAMOS, G., and YALAMANCHILI, S., "A characterization and analysis of ptx kernels," in *IISWC09: IEEE International Symposium on Workload Characterization*, (Austin, TX, USA), October 2009.

[96] KERR, A., DIAMOS, G., and YALAMANCHILI, S., "Modeling gpu-cpu workloads and systems," in *Third Workshop on General-Purpose Computation on Graphics Procesing Units*, (Pittsburg, PA, USA), March 2010.

[97] KERR, A., DIAMOS, G., and YALAMANCHILI, S., "Gpu application development, debugging, and performance tuning with gpu ocelot.," in *GPU Computing GEMS , vol. 2* (WEN-MEI HWU, E. A., ed.), ch. 30, Morgan Kaufmann, 2011.

[98] KERR, A., DIAMOS, G., and YALAMANCHILI, S., "Dynamic compilation of data-parallel kernels for vector processors," in *Proceedings of the 2012 10th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2012, 2012.

[99] KHAN, M. A., "Improving performance through deep value profiling and specialization with code transformation," *Comput. Lang. Syst. Struct.*, vol. 37, pp. 193–203, Oct. 2011.

[100] KHAN, M. A., CHARLES, H. P., and BARTHOU, D., "Languages and compilers for parallel computing," ch. An Effective Automated Approach to Specialization of Code, pp. 308–322, Berlin, Heidelberg: Springer-Verlag, 2008.

[101] KIRK, D., "Nvidia cuda software and gpu parallel computing architecture," in *ISMM '07: Proceedings of the 6th international symposium on Memory management*, (New York, NY, USA), pp. 103–104, ACM, 2007.

[102] KOTHA, A., ANAND, K., SMITHSON, M., YELLAREDDY, G., and BARUA, R., "Automatic parallelization in a binary rewriter," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 547–557, IEEE Computer Society, 2010.

[103] LABS, I., "The single-chip cloud computing platform overview," tech. rep., Intel Corp., 2009.

[104] LAINE, S. and KARRAS, T., "High-performance software rasterization on gpus," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, (New York, NY, USA), pp. 79–88, ACM, 2011.

[105] LATTNER, C. and ADVE, V., "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO '04: Proceedings of the international symposium on Code generation and optimization*, (Washington, DC, USA), p. 75, IEEE Computer Society, 2004.

[106] LEE, J., KIM, J., SEO, S., KIM, S., PARK, J., KIM, H., DAO, T. T., CHO, Y., SEO, S. J., LEE, S. H., CHO, S. M., SONG, H. J., SUH, S.-B., and CHOI, J.-D., "An opencl framework for heterogeneous multicores with local memory," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 193–204, ACM, 2010.

[107] LEE, J. and KIM, H., "Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2012.

[108] LEE, S., GROVER, V., CHAKRAVARTY, M. M. T., and KELLER, G., "Gpu kernels as data-parallel array computations in haskell," 2009.

[109] LEE, S., MIN, S.-J., and EIGENMANN, R., "Openmp to gpgpu: a compiler framework for automatic translation and optimization," *SIGPLAN Not.*, vol. 44, pp. 101–110, February 2009.

[110] LEISA, R., HACK, S., and WALD, I., "Extending a c-like language for portable simd programming," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 65–74, ACM, 2012.

[111] LI, W. and PINGALI, K., "Access normalization: Loop restructuring for numa computers," in *ACM Transactions on Computer Systems*, 1993.

[112] LUK, C., HONG, S., and KIM, H., "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO'09*, (New York, USA), IEEE, devember 2009.

[113] MALEKI, S., GAO, Y., GARZARÁN, M. J., WONG, T., and PADUA, D. A., "An evaluation of vectorizing compilers," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, (Washington, DC, USA), pp. 372–382, IEEE Computer Society, 2011.

[114] MANLY, B. F., *Multivariate statistical methods: a primer*. London, UK, UK: Chapman & Hall, Ltd., 1986.

[115] MEIJER, E., WA, R., and GOUGH, J., "Technical overview of the common language runtime," 2000. http://research.microsoft.com/emeijer/Papers/CLR.pdf.

[116] MERRILL, D., *Allocation-oriented Algorithm Design Allocation oriented with Application to GPU Computing*. PhD thesis, University of Virginia, 2011.

[117] MERRILL, D. G. and GRIMSHAW, A. S., "Revisiting sorting for gpgpu stream architectures," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 545–546, ACM, 2010.

[118] MERRITT, A. M., GUPTA, V., VERMA, A., GAVRILOVSKA, A., and SCHWAN, K., "Shadowfax: scaling in heterogeneous cluster systems via gpgpu assemblies," in *Proceedings of the 5th international workshop on Virtualization technologies in distributed computing*, VTDC '11, (New York, NY, USA), pp. 3–10, ACM, 2011.

[119] MUCHNICK, S., *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[120] NEWBURN, C., SO, B., LIU, Z., MCCOOL, M., GHULOUM, A., TOIT, S., WANG, Z. G., DU, Z. H., CHEN, Y., WU, G., GUO, P., LIU, Z., and ZHANG, D., "Intel's array building blocks: A retargetable, dynamic compiler and embedded language," in *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pp. 224 –235, april 2011.

[121] NORM RUBIN, A., "Hsail: an introduction to the hsa intermediate language," tech. rep., 2012.

[122] NVIDIA, *CUDA CUBLAS Library*. NVIDIA Corporation, Santa Clara, California, September 2008.

[123] NVIDIA, *NVIDIA Compute PTX: Parallel Thread Execution*. NVIDIA Corporation, Santa Clara, California, 1.3 ed., October 2008.

[124] NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture*. NVIDIA Corporation, Santa Clara, California, 2.1 ed., October 2008.

[125] NVIDIA, "Nvidias next generation cuda compute architecture: Fermi," tech. rep., NVIDIA Coporation, 2009.

[126] NVIDIA, *NVIDIA CUDA Library Documentation*. NVIDIA Corporation, Santa Clara, California, 3.2 ed., October 2010.

[127] NVIDIA, "Nvidia geforce gtx 680: the fastest, most efficient gpu ever built," tech. rep., 2011.

[128] PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., and STICH, M., "Optix: a general purpose ray tracing engine," *ACM Trans. Graph.*, vol. 29, pp. 66:1–66:13, July 2010.

[129] REAÑO, C., PEÑA, A. J., SILLA, F., MAYO, R., QUINTANA-ORTÍ, E. S., and DUATO, J., "CU2rCU: towards the Complete rCUDA Remote GPU Virtualization and Sharing Solution," in *19th Annual International Conference on High Performance Computing (HiPC)*, December 2012.

[130] RINARD, M. and DINIZ, P., "Commutativity analysis: A new analysis framework for parallelizing compilers," tech. rep., Santa Barbara, CA, USA, 1996.

[131] RYOO, S., UENG, S.-Z., RODRIGUES, C. I., KIDD, R. E., FRANK, M. I., and HWU, W.-M. W., "Transactions on high-performance embedded architectures and compilers i," ch. Automatic Discovery of Coarse-Grained Parallelism in Media Applications, pp. 194–213, Berlin, Heidelberg: Springer-Verlag, 2007.

[132] SAMEH, A. H. and KUCK, D. J., *On Stable Parallel Linear System Solvers*. Journal of the ACM, 1978.

[133] SAMET, H., *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, first ed., 2006.

[134] SAMPAIO, D., MARTINS, R., COLLANGE, S., and MAGNO QUINT
AO PEREIRA, F., "Divergence Analysis with Affine Constraints," tech. rep., Nov. 2011.

[135] SCHWARTZ, D., JUDD, R., HARROD, W., and MANLEY, D., *VSIPL 1.3 API*. VSIPL Forum, January 2008.

[136] SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., and HANRAHAN, P., "Larrabee: a many-core x86 architecture for visual computing," in *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, (New York, NY, USA), pp. 18:1–18:15, ACM, 2008.

[137] SHIN, J., "Introducing control flow into vectorized code," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, (Washington, DC, USA), pp. 280–291, IEEE Computer Society, 2007.

[138] SMITH, M. D., "Overcoming the challenges to feedback-directed optimization (keynote talk)," in *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, DYNAMO '00, (New York, NY, USA), pp. 1–11, ACM, 2000.

[139] SMITH, M. D., HOROWITZ, M., and LAM, M. S., "Efficient superscalar performance through boosting," *SIGPLAN Not.*, vol. 27, pp. 248–259, September 1992.

[140] STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., and MOWRY, T. C., "A scalable approach to thread-level speculation," in *IN PROCEEDINGS OF THE 27TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE*, pp. 1–12, 2000.

[141] STEFFEN, M. and ZAMBRENO, J., "Improving simt efficiency of global rendering algorithms with architectural support for dynamic micro-kernels," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 237–248, IEEE Computer Society, 2010.

[142] STRATTON, J., GROVER, V., MARATHE, J., AARTS, B., MURPHY, M., HU, Z., and MEI HWU, W., "Efficient compilation of fine-grained spmd-threaded programs for multicore cpus," in *CGO 2010*, (Toronto, Canada), April 2010.

[143] STRATTON, J., GROVER, V., MARATHE, J., AARTS, B., MURPHY, M., HU, Z., and MEI HWU, W., "Efficient compilation of fine-grained spmd-threaded programs for multicore cpus," in *CGO 2010*, (Toronto, Canada), April 2010.

[144] STRATTON, J., STONE, S., and MEI HWU, W., "Mcuda: An efficient implementation of cuda kernels on multi-cores," Tech. Rep. IMPACT-08-01, University of Illinois at Urbana-Champaign, March 2008.

[145] SUGANUMA, T., YASUE, T., and NAKATANI, T., "A region-based compilation technique for dynamic compilers," *ACM Trans. Program. Lang. Syst.*, vol. 28, pp. 134–174, January 2006.

[146] SUGANUMA, T., YASUE, T., and NAKATANI, T., "A region-based compilation technique for dynamic compilers," *ACM Trans. Program. Lang. Syst.*, vol. 28, pp. 134–174, Jan. 2006.

[147] TARJAN, D., BOYER, M., and SKADRON, K., "Federation: repurposing scalar cores for out-of-order instruction issue," in *Proceedings of the 45th annual Design Automation Conference*, DAC '08, (New York, NY, USA), pp. 772–775, ACM, 2008.

[148] THIES, W., CHANDRASEKHAR, V., and AMARASINGHE, S., "A practical approach to exploiting coarse-grained pipeline parallelism in c programs," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, (Washington, DC, USA), pp. 356–369, IEEE Computer Society, 2007.

[149] TIAN, C., FENG, M., NAGARAJAN, VIJAY, and GUPTA, R., "Copy or discard execution model for speculative parallelization on multicores," in *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 330–341, IEEE Computer Society, 2008.

[150] TOMOV, S., DONGARRA, J., and BABOULIN, M., "Towards dense linear algebra for hybrid gpu accelerated manycore system," October 2008.

[151] VALIANT, L. G., "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[152] VASILY, V. and W., D. J., "Benchmarking gpus to tune dense linear algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, (Piscataway, NJ, USA), pp. 1–11, IEEE Press, 2008.

[153] VASILY, V. and W., D. J., "Benchmarking gpus to tune dense linear algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, (Piscataway, NJ, USA), pp. 1–11, IEEE Press, 2008.

[154] WU, H., DIAMOS, G., LI, S., and YALAMANCHILI, S., "Characterization and transformation of unstructured control flow in gpu applications," in *First International Workshop on Characterizing Applications for Heterogeneous Exascale Systems*, June 2011.

[155] WU, H., DIAMOS, G., LELE, A., WANG, J., CADAMBI, S., YALAMANCHILI, S., and CHAKRADHAR, S., "Optimizing data warehousing applications for gpus using kernel fusion/fission," in *Multicore and GPU Programming Models, Languages and Compilers Workshop*, May 2012.

[156] YARDIMCI, E. and FRANZ, M., "Dynamic parallelization and mapping of binary executables on hierarchical platforms," in *Proceedings of the 3rd conference on Computing frontiers*, CF '06, (New York, NY, USA), pp. 127–138, ACM, 2006.

[157] YARDIMCI, E. and FRANZ, M., "Mostly static program partitioning of binary executables," *ACM Trans. Program. Lang. Syst.*, vol. 31, pp. 17:1–17:46, July 2009.

[158] ZENG, H. and GHOSE, K., "Register file caching for energy efficiency," in *Proceedings of the 2006 international symposium on Low power electronics and design*, ISLPED '06, (New York, NY, USA), pp. 244–249, ACM, 2006.

[159] ZHANG, E. Z., JIANG, Y., GUO, Z., TIAN, K., and SHEN, X., "On-the-fly elimination of dynamic irregularities for gpu computing," *SIGPLAN Not.*, vol. 46, pp. 369–380, March 2011.

[160] ZHANG, X., WANG, Z., GLOY, N., CHEN, J. B., and SMITH, M. D., "System support for automatic profiling and optimization," *SIGOPS Oper. Syst. Rev.*, vol. 31, pp. 15–26, October 1997.